USENIX

C++ CONFERENCE PROCEEDINGS

SPRING

1994

C++

CONFERENCE PROCEEDINGS

Cambridge, MA
April 11-14, 1994

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $24 for members and $28 for nonmembers.
Outside the U.S.A. and Canada, please add
$20 per copy for postage (via air printed matter).

Past USENIX C++ Conferences

C++ Conference	August 1992	Portland, OR	$30/39
C++ Conference	April 1991	Washington, DC	$22/26
C++ Conference	April 1990	San Francisco, CA	$28
C++ Conference	October 1988	Denver, CO	$30
C++ Workshop	November 1987	Santa Fe, NM	$30

ISBN 1-880446-60-X

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the 1994 USENIX

C++ Conference

April 11-14, 1994
Cambridge, MA, USA

TABLE OF CONTENTS

USENIX Sixth C++ Technical Conference
April 11 -14, 1994
Cambridge, Massachusetts

Program Committee

Doug Lea, Program Chair, *State University of New York at Oswego*
Desmond D'Souza, *Icon Computing*
Erich Gamma, *Taligent*
Judith Grass, *Corporation for National Research Initiatives*
Mark Linton, *Silicon Graphics, Inc.*
Scott Meyers, *Consultant*
Vince Russo, *Purdue University*
Michael Tiemann, *Cygnus Support*
Steve Vinoski, *Hewlett-Packard*
Jim Waldo, *Sun Microsystems Laboratories*

The Object Binary Interface — C++ Objects for Evolvable Shared Class Libraries

Theodore C. Goldstein
Sun Microsystems Laboratories
ted.goldstein@eng.sun.com

Alan D. Sloane
SunPro
alan.sloane@eng.sun.com

Abstract

Object-oriented design and object-oriented languages support the development of independent software components such as class libraries. When using such components, versioning becomes a key issue. While various ad-hoc techniques and coding idioms have been used to provide versioning, all of these techniques have deficiencies - ambiguity, the necessity of recompilation or re-coding, or the loss of binary compatibility of programs. Components from different software vendors are versioned at different times. Maintaining compatibility between versions must be consciously engineered. New technologies such as distributed objects further complicate libraries by requiring multiple implementations of a type simultaneously in a program.

This paper describes a new C++ object model called the *Shared Object Model* for C++ users and a new implementation model called the *Object Binary Interface* for C++ implementors.These techniques provide a mechanism for allowing multiple implementations of an object in a program. Early analysis of this approach has shown it to have performance broadly comparable to conventional implementations.

1 Introduction

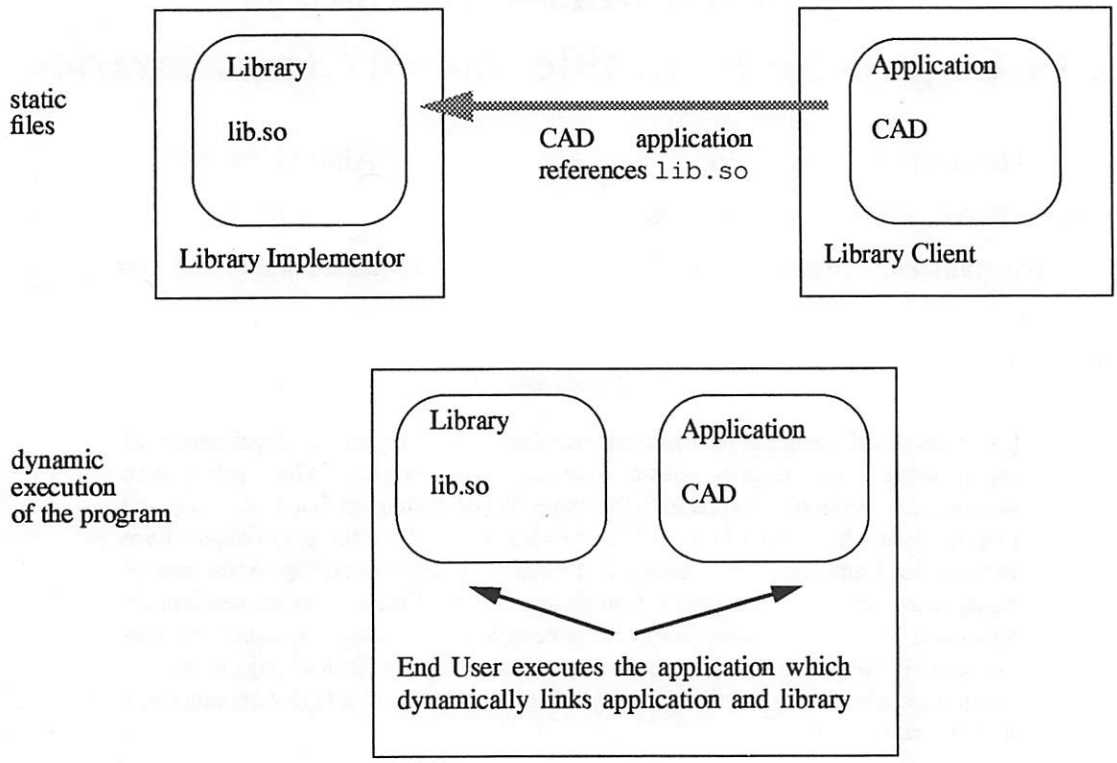
Software either evolves or dies. Software evolution occurs in response to numerous requirements including bug fixes, user demands for greater functionality, and especially to support changes in related software. Object-oriented programming promises to allow individual class library components to evolve independently of its clients. Many modern operating systems allow software libraries to be efficiently shared among individual program address spaces using dynamically linked shared libraries. Shared libraries work by deferring certain binding operations until the program is loaded and executed. This sharing provides efficient utilization of the computer systems memory, but deferring the binding time of class libraries introduces the risk of version incompatibility between library and client software. But deferring the binding time also provides the opportunity to improve software by introducing new functionality and fixing software defects.

This work builds upon the idea of evolvable classes introduced by [Ellis & Stroustrup] suggesting the use of tables of offsets to members. Andrew Palay further developed the ideas of evolvable classes in his ΔC++ system, [Palay]. Like ΔC++, we define certain compatible changes to a class library that will not require recompilation or changes to a client.

1.1 Compatible Evolution

The following example illustrates the relationship between the implementor of a C++ library, the developer of an application (or another library) who uses that library, and the end-user of the application.

FIGURE 1. Library, Application and End-User



Suppose the library implementor changes the implementation of `lib.so` or changes the interface in an upwardly compatible way to provide additional functionality and ships the new version to the end-user. The end-user expects - quite reasonably - that the expensive CAD package bought from the ISV will still run. The end-user cannot recompile the CAD package - she doesn't have sources, and may not even have a compiler. On the other hand the library-implementor, even if it's the platform library implementor, can't require all ISV's to synchronize with them. In reality, synchronization will be even more complicated since the CAD package likely requires libraries from several different vendors. To fulfill the promise of modular software components, object-oriented technology must support compatible replacement of software libraries.

It must be possible to introduce compatible changes to a library without requiring any changes to the source code or even recompilation of the clients of the library. Run-time mechanisms for object-oriented languages must support *compatible evolution* of class libraries.

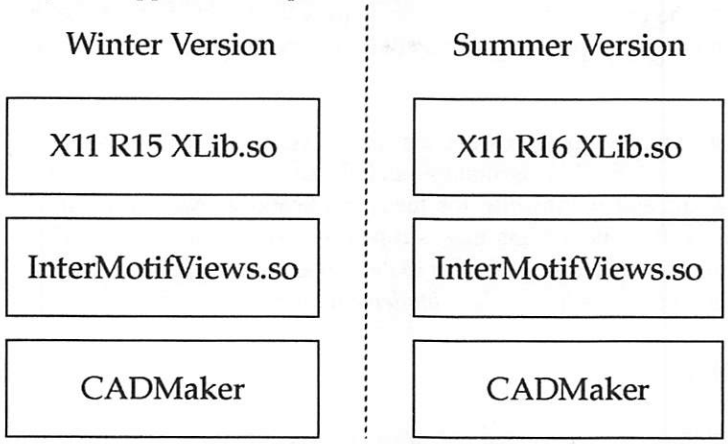
1.2 Interface-Implementation Independence

Recent technologies such as the Object Management Group's CORBA [OMG] which support distributed applications allow objects to span address spaces. Objects may be defined in one program and used in another. Objects move from one address space to another through a linearization and communication process such as *remote procedure call* [Birrell & Nelson] or Object Invocation [OMG]. A very useful mechanism is for the external address space to pass an instance of a derived class. Frequently, during the unmarshalling of the object, it may be discovered that the new address space does not have the derived class implementation of the object, but only have the base class interface of the object. Dynamic linking provides a mechanism a program may use to acquire the corresponding implementation for a marshalled object.

Imagine, for example, that X11's *XLib* library is written in C++. Suppose there is a hypothetical third party library vendor writing a window system toolkit called *InterMotifViews* that uses XLib. Among the

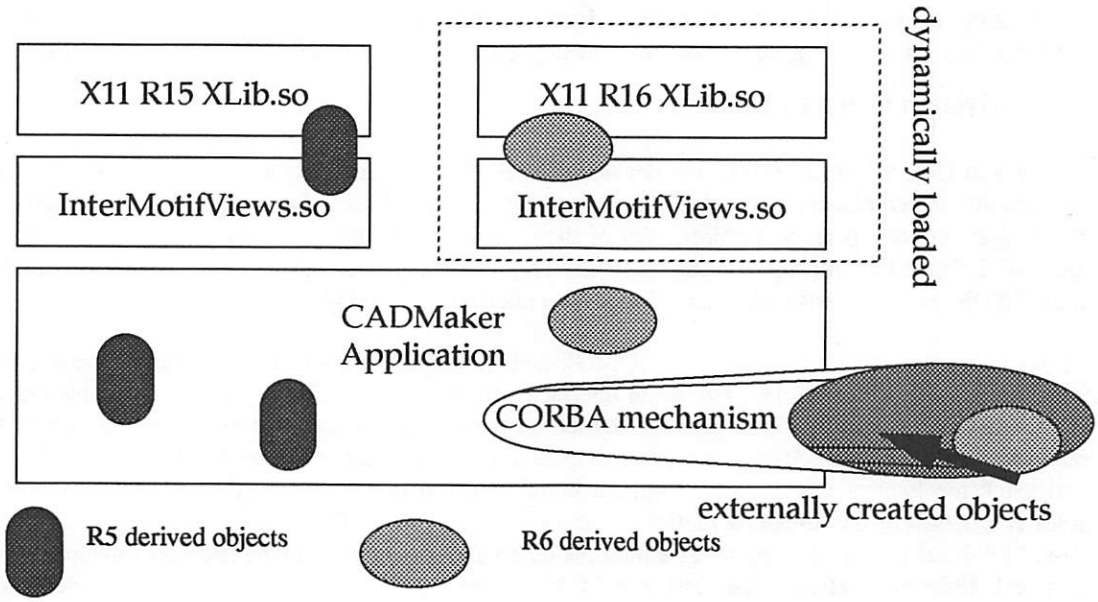
applications using InterMotifViews is a sophisticated network oriented application called *CADMaker*. Since XLib and InterMotifViews come from separate vendors, their will always be a time delay between release of the XLib library and the next release of InterMotifViews. It may even be possible that end-users receive updates of the X11 library before the maker of InterMotifViews sees it. Figure 2 depicts a Winter and following Summer rconfiguration of the CADMaker application.

FIGURE 2. Library and Application Dependencies



Suppose the environment and libraries supports multiple user distributed applications. The hypothetical InterMotifViews is designed for running workgroup applications which allow users to work together. As well, some servers at a CADMaker client site are running the Winter version and some are running the Summer version of XLib. CADmaker programs share objects. Objects which are made by compatible but different class libraries in an external program are transferred and shared from one address space to another through mechanisms such as CORBA. Figure 3 depicts the summertime execution context of CADMaker. where X11 R16 implementation objects come from an external address space marshalled through some RPC mechanism.

FIGURE 3. Marshalled objects



The correct behavior is that the application has access to both R15 and R16 libraries simultaneously. If XLib objects are accessed only through virtual functions, the CADMaker application should be unaware of the

version differences between XLib R15 and XLib R16 created objects. The code mechanisms of unmarshalling (and dynamic loading) arrange to get the correct addresses for the new library into the virtual function tables.

A single CADMaker application will contain instances derived from both R15 and R16 XLib code. Enabling such multiple versions can be easily achieved by strictly respecting the encapsulation boundary of objects. An object should not access the private data of any other object, even those of its own class, except through virtual function calls. The private data of an object is truly kept private. Thus the only restriction over the normal C++ object model is that private data may only be accessed either explicitly or implicitly through the "this pointer."

These usage rules place restrictions on what the library vendor can do with C++ objects. Despite these restrictions, the advantages of allowing multiple definitions of an object in the same address space at the same time is significant and worthwhile for many applications. Note also that even in ordinary non-distributed applications dynamic loading makes it possible to have multiple implementations of an object coexist in the same address space.We will refer to the property of supporting multiple implementations of a type within the one address space as *interface-implementation independence*.

2 Related work

Several previous authors have discussed the issue of versions and evolution, some in the context of programming languages, but most in relation to schema evolution in object-oriented databases. A brief summary of these approaches is described in [Goldstein].

2.1 ΔC++

ΔC++'s solution does not support multiple implementations of an object in an address simultaneously. ΔC++ allows a developer to make compatible changes to a class with minimal recompilation of clients. The set of compatible changes maintains or extends the class interface without altering the code sequence used to access members of that class. ΔC++ accomplishes this by resolving classes at link time. Changes are handled by extending the linker to support new relocation types such as member offset. Most code sequences are identical to `cfront` [Ellis & Stroustrup] generated code except for constructors/destructors, calls to non-virtual members, access to embedded structures and some optimizations. The expense of this technique is additional program start-up time latency while the linker processes the additional relocations.

2.2 System Object Model

The System Object Model (SOM) was developed at IBM. There are several variants - SOM, DSOM that supports distributed objects within the general framework of the Object Management Group's CORBA, and PSOM that supports persistent objects. All of these provide run-time support for a common interface to "objects" independent of programming language. They also have an extensive metaclass framework, which in the DSOM case supports the dynamic invocation interface of CORBA.

By extending IDL's notion of interfaces, [OMG], with an implementation construct called "release-order:" SOM supports extension and change of an interface. Other aspects of class evolution are implemented by SOM's metaclass protocol. SOM puts the burden on the user to maintain the correct release order across versions. For example, the release-order entries must only be extended, never removed or deleted. SOM uses a dispatch mechanism that supports multiple inheritance but favors single inheritance implementations. SOM is a hybrid of the Smalltalk metaclass object protocol and the OMG object model layered on top of C++. Additional overhead is required for multiple inheritance involving caching and hash lookup tables. The principal difference between the OBI and SOM is that the OBI approach still assumes the essential mechanism and efficiency found in C++, while SOM emulates the object model using Smalltalk-like framework.

2.3 Schema Evolution

The object-database community has long recognized the necessity for supporting type and schema evolution. Object-database systems that support evolution include: Orion [Bannerjee], GemStone [Penney & Stein] and O2 [Bancilhon]. The E compiler in the Exodus system [Richardson and Carey] follows a technique similar to the OBI in using offsets to point to virtual base classes. But no support for multiple versions of an object class in a database has been described in this work.

2.4 Design focus of the OBI

All the mechanisms that have been proposed to support evolution of class definitions either involve significant extra indirection at run-time or require extra relocation work by the runtime linker. Linker start-up latency is already noticeable in dynimcally-linked C++ programs. Our concern is that additional start-up latency would be unacceptable.

Unlike ΔC++ and SOM we believe that not all changes of a class interface are of equal utility between version release. During software development, radical rearrangement of function order and other class hierarchy is useful. There is little value in supporting arbitrary reordering of member functions between two *released* versions of a class. Our selections of changes and usage rules provide compatible evolution of types and interface-implementation independence with limited impact on C++ usage. The OBI design highest priority is to allow the private parts of an object to change arbitrarily. The second highest priority is to allow extension by addition [Harrison & Ossher]. Other changes such as rearrangement of class hierarchies are not unimportant, but are not the focus of this work. If desired, it is simple to add a #pragma annotation which specifies the order of data members. The OBI design differs from previous work in that it focuses on a select few high values changes and it supports multiple simultaneous implementations of a class within an address space.

3 Existing Support of versioning in C++

There are several techniques that might support versioning within C++ including renaming, derivation, and namespaces. All of these techniques provide some handle on the problem, but all are insufficient to meet our criteria stated above. Renaming functions, for example has been around as long there have been symbolic identifiers. Version 6 UNIX had a file seeking operating system call named `seek` which took one 16 bit integer argument. The obvious flaw of files exceeding 64 K was fixed in version 7 UNIX by adding in the operating system call named `lseek`. This works amazingly well because both the old and new operations can coexist. In practice there is no limit to the number of "l"'s or other version-specific characters one can add onto an identifier. But exceedingly long identifiers become unwieldy and are a blight on the code.

Objects require additional support. Users would like new objects to be accepted anywhere an old object was accepted. One solution, proposed in [Hamilton & Radia], is to represent version conformance explicitly by modeling it using derivation.

```
class UNIX_File_v6 {
      unsigned short seek(unsigned short offset, int whence);
};
class UNIX_File_v7 : public UNIX_File_v6 {
      unsigned long seek(unsigned long offset, int whence);
};
```

As an alternative the recently defined namespace extension [Stroustrup] to the ANSI/ISO C++ draft standard provides a mechanism that can alleviate the unwieldiness of the long identifiers. Applications that contain objects derived from the UNIX_File_v6 class must be edited and recompiled to take advantage of the newer UNIX_File_v7 class. We want a solution that allows application objects to transparently use the latest version of the class. The approach we have chosen makes versions orthogonal to the C++ type system.

4 The Shared Object Model

The Shared Object Model provides a C++ programmer with an additional annotation to partition class library into classes that the implementor guarantees never to change, and classes that may change. The first category often corresponds to "key" components of the library whose interface is extremely well understood and for which optimal implementations are well known. The overall run-time performance of the library is dominated by the performance of these key components. For example, an implementor of a reference counting object might decide to exposure via inline functions the implementation. The implementor is trading off the risk of the need to change the implementation of the reference counting object in order to gain better performance. Inlines effectively exports the implementation (e.g. the layout of the private members) of a class and so precludes any possibility of evolution.

The second category corresponds to outer layers of abstraction, usually having more complex semantics and larger granularity. Thus adding overhead to function calls for these layers will have less impact on the performance of the library as a whole. Correspondingly these are the layers whose semantics are most subject to change from version to version and where there is greater variability in implementation. They are the classes where support for evolution is most important. Support for evolution is not appropriate for all classes, but where library design makes a natural partitioning of classes such that evolution is both possible and effective for critical classes.

Recognizing this, we chose to add extra syntax to identify evolvable classes to the compiler. We considered using a `#pragma`, but instead chose to use a linkage specification: `extern "shared" { ... }`. A linkage specification fits very well with our requirements:

(i) It is not a language extension. We abhor any more extensions to this language. But the C++ draft standard allows an implementation to provide an arbitrary set of linkage specifications in addition to the required specifications `"C"` and `"C++"`. We have defined a specification `"shared"`, which results in linkage conventions suitable for long-lived interfaces and for linkage across shared object boundaries.

(ii) Code using linkage specifications is portable. Compilers which do not support a particular linkage specification will generate a warning, but supply default linkage.

(iii) It is semantically accurate. The concept of "linkage" is concerned with communication across translation unit and library boundaries, especially if it provides cross-language models.

(iv) It is syntactically appropriate. Entire header files can be easily "wrapped" to use shared linkage, just as existing C header files could be given C linkage with `extern "C"`.

4.1 extern "shared" { ... }

We chose the word *shared* because we wanted to allude to System V shared libraries (dynamically linked libraries), and because most of the suggested alternatives, such as `dynamic`, `global` and `export`, had other conflicting connotations. In our implementation a `"shared"` linkage specification affects only class definitions and their member functions and does not affect non-member functions. The layout of class objects and the mechanism for calling virtual functions depends on the *linkage specification* of the class. For example in the following code fragment:

```
extern "C++" {
        void f(int);
        class A { ... };
        class B : public A { ... };
};
class C { ... };
extern "shared" {
        int g(X*);
        class X { ... };
        class Y : public X { ... };
};
```

The function `void f(int)` and the classes `A`, `B` and `C` have default or "C++" linkage; and the function g(X*) and the classes `X`, `and` `Y` have "`shared`" linkage. A given class can have only one linkage specification throughout a program (This is the same rule which applies to the linkage specification of functions in the C++ draft standard). Moreover the representation of a *pointer-to-member-of-shared-class* is different to the representation of a *pointer-to-member-of-default-class* and assignment of one to the other is not allowed. In addition there is a semantic restriction on how classes with different linkage specifications can be combined. A derived class and all its base classes must have the same linkage specification.

4.2 Semantics of the Shared Object Model

The principle new rule of the OBI is that the private data of an object must be strictly encapsulated. Thus the 1990's corollary of "only friends may touch your private parts is that "*no one else may touch your private parts*." The official rules of the OBI are:

(i) The only access to the private data members of an object is through the object's (explicit or implicit) `this` pointer. Non-virtual functions, both member functions and non-member friend functions, cannot access the private parts. The shared object model does not prohibit access to public or protected data members.

(ii) New public and protected members (both data-members and virtual functions) must be added to the end of the list of existing members.

(iii) The one definition rule for object types is relaxed for shared linkage types within a program (but not within a single compilation unit).

5 The OBI implementation model

The shared object model is implemented using the Object Binary Interface (OBI) implementation model. Of course, users do not need to know the mechanics of the OBI implementation model. The principle notion is that the OBI is a binary interface similar in spirit to the System V Application Binary Interface (ABI). It is possible that a future generation of the OBI may allow for binary compatible linkage of other Object-oriented languages besides C++. The OBI has three basic concepts:

(i) The OBI specifies that each instance of a class with shared linkage contains an `optr` (pronounced oh-pointer) as its first element. The `optr` points to an `otbl` which describes the layout of the class instance and how to invoke the virtual functions of the class.

(ii) Public and protected data members come first. After all public and protected data members, come the private data members. Consequently the public and protected data members of a class can be extended because only the private data is adjusted. But the total ordering of the public and protected data members is preserved!

(iii) All inheritance is implemented in a fashion similar to virtual inheritance. The fundamental mechanism is to allow the relationship between base and derived objects to change and evolve. The virtual inheritance mechanisms in `cfront` provide the inspiration for the necessary indirection.

As with many simple ideas, these two rules generate many important implementation details. The rest of this paper describes the OBI implementation model, including several key algorithms.

5.1 Otbls

For classes with shared linkage, `otbls` take the place of `vtbls`. The terms `optr` and `otbl` originate from the `vptr` and `vtbl` described in [Ellis & Stroustrup]. The 'o' may stand for 'offset' or 'object'. Within an object, the public and protected data members immediately follow the `optr`. The object's `optr` points to the `size` field in the fixed part of the `otbl`. The `base_part` structures are indexed backwards from there, and `function_part` structures are indexed forwards. The `base_part` structures are indexed in order on a left-to-right, pre-order depth-first traversal of the inheritance DAG, with `base_part`'s for virtual bases left to the end and allocated in the order the virtual bases were encountered in the graph traversal. The `function_part` structures are allocated for each virtual function defined in the class in order of declaration.

An `otbl` is global data, and will have an external "mangled" name. It can be treated as a global variable or (probably better) as a private static data member of its class. One difference between `vtbls` and `otbls`, is that `vtbls` along the left linear tree walk of the derivation are concatenated. This concatenation is not possible with `otbls`, as it would prohibit growth in adding new virtual functions to the end.

5.2 Class Layout

Within the public/protected section and the private section the members are laid out in declaration order. The order in which the base classes are laid out is unimportant, since all accesses to inherited members are calculated through offsets found in the `otbl`. The `otbl` consists of three parts:

(i) an array of `base_part` structures which grows backwards providing base class information. Each `base_part` consists of a pointer to an `otbl` and the offset of the base class object

```
struct __otbl{
    struct base_part {
        __otbl* base;
        size_t widen_offset;
    } base[n_bases];
```

(ii) a fixed descriptor section describing the size of instances, the number of direct virtual functions, and the total number of base classes. This part also contains access to the runtime type information.

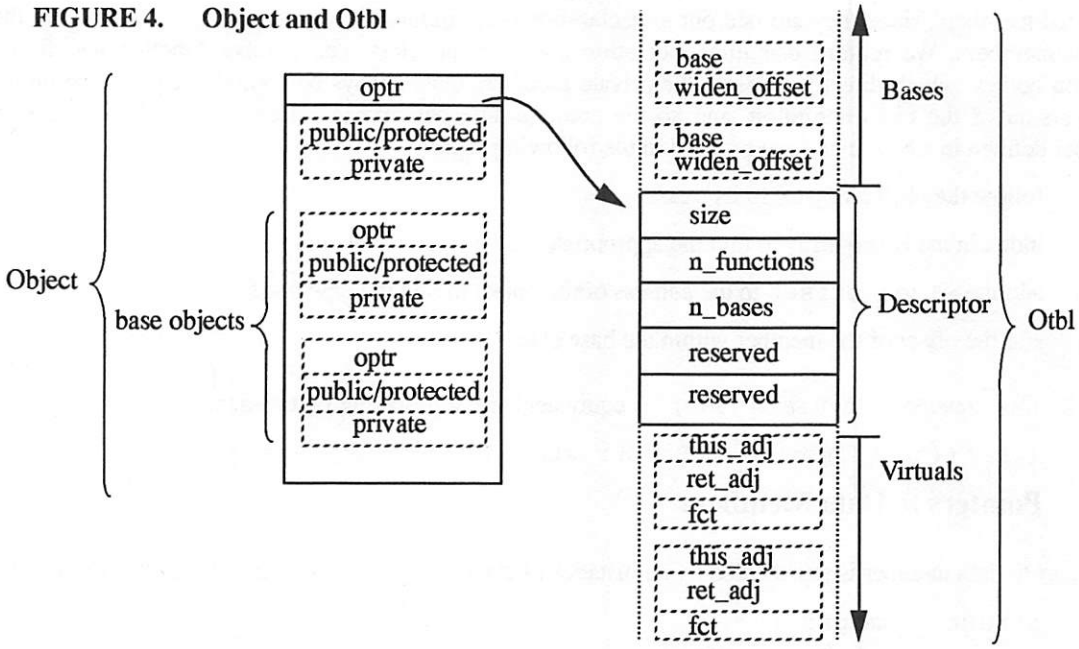
```
    size_t size;
    size_t n_functions;
    size_t n_bases;
    TypeInfo* tinfo;
    size_t most_widen_offset;
```

(iii) an array of `function_part` structures which consist of a pointer to a direct virtual function of the class and adjustments for the object pointer and the return value.

```
struct function_part {
        size_t this_adj;
        size_t ret_adj;
        void (*fct)();
    } function[n_functions];
```

The relationship between an object and its otbl is presented in Figure 4 on the next page. The otbl points to the "otbl descriptor" which is located in the middle of the object. In the following sections we show through pseudo-code and diagrams how classes with shared linkage and objects of such classes are used. This description covers *all* possible uses of the object by client code.

FIGURE 4. Object and Otbl



5.3 Allocation of automatic and static instances

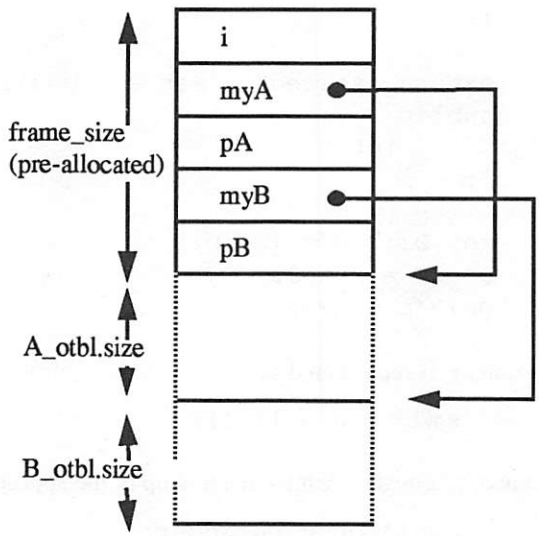
Automatic objects of a class with shared linkage are allocated on the stack frame via an `alloca`-like mechanism detailed below. Objects are represented by a pointer to a location elsewhere in the stack frame and the "real" object is laid out by special prologue code in the function. Static objects, whether global or file static, are represented by a pointer and the "real" object is on the heap since the real object is not known. Allocation and initialization of statics are done (as is conventional) with `.init` code.

FIGURE 5. Allocation of auto objects in Shared Linkage

```
extern "shared" {
      class A {
      public:
            int a1;
            int a2;
      };

      class B : public A {
      public:
            int b;
      };
}

void func() {
      int i;
      A myA, *pA;
      B myB, *pB;
}
```



5.4 Access to Data Members

Just as in default linkage, to access a member defined in the class itself (not a base class) we use an offset determined at compile-time. Notice that changing the size of the private part or extending the list of public and protected members does not change the offsets compiled into client code for existing public and

protected members, since they are laid out in declaration order immediately after the `optr` and before the private members. We require that implementation code for the class, i.e. member function and friend function bodies, which directly accesses the private members must always be compiled against the up-to-date version of the class definition, and so the compile-time determined offsets are valid. To access a member defined in a base class[1] we proceed in the following steps

(i) follow the object's `optr` to its `otbl`

(ii) index in the `base` array to find the appropriate `base_part` structure

(iii) add the `widen_offset` to the address of the object to find the base class object

(iv) add the offset of the member within the base class.

Thus the C++ statement "`myB.a2= 999;`" is equivalent to the following C statement

```
((A*)((char*)&myB + myB.optr->base[0].offset))->a2= 999;
```

5.5 Pointers to Data Members

A pointer to data member is represented by an instance of class `__smdptr`, which is defined as follows:

```
struct __smdptr {
        size_t base_index;
        size_t offset;
};
```

The `base_index` is negative if the it a direct member, and not a member of a base class. An example of accessing a pointer-to-data-member is:

```
extern "shared" class B {
public:
        int b1:
        int b2;
};

extern "shared" class D : public B {
public:
        int d:
};

int D::* p = &D::b2;
D* pD = new D;
pD->*p = 999;
```

The pointer `p` is represented as

```
__smdptr p = {1,4};
```

Referencing through `p` results in a lookup of the appropriate `base_part` structure in the `otbl`

```
__otbl* ot = pD->optr;
char* pTmp =(B*)((char*)pD + (p.base_index < 0 ? 0 :
                    ot->base[p.base_index].widen_offset));
* ( (int*) ( pTmp + (char*)p.offset ) ) = 999;
```

---

1. In client code this must be a public or protected member of a public or protected base

5.6 Derivation and Casts

Casts are implemented much as in conventional implementations, see for example [Ellis & Stroustrup] pages 221-227, except that the adjustments applied to pointers are not compile-time constants, but rather must be looked up in the object's `otbl`.

Downcasts (i.e. casts from base to derived class) are implemented by casting to the most derived class using the `most_widen` field, and then back to the intermediate class. As a consequence of this, downcasts from virtual bases are supported, but the result of downcasting to a non-virtual intermediate base which occurs twice in the hierarchy is undefined. This is the only difference we have identified between shared linkage and "standard" C++, and many users find it less restrictive than the absence of casts from virtual bases. In passing we note that this same restriction is also present in SOM's C++ mapping.

FIGURE 6. Cast from a derived class to a base class

```
extern "shared" {
        class V { public: int v; private: ... };
        class A : public virtual V {
                ...
        public:
                int a:
        private:
                ...
        };
        class B : public virtual V {
        public:
                int b;
        private:
                ...
        };
        class C : public A, public B {
        public:
                int c;
        private:
                ...
        };
}

C* pC = new C;
B* pB = pC;// implicit cast, B* ← C*
V* pV = pB; // implicit cast, V* ← B*
```

The layout of the object pointed to by `pC`, and it's network of `otbls` is shown in Figure 9 below. Pseudo-code illustrating the implementation of the casts is

```
__otbl* ot = pC->optr;
pB = (B*) ((char*)pC + ot->base[1].offset);
__otbl* ot2 = pB->optr;// B in C otbl
pV = (V*) ((char*)pB + ot2->base[0].offset)
```

Unlike normal vtbls, there is no difference in the code generated for casts to virtual and non-virtual bases[1].In the example,

```
__otbl* ot = pC->optr;
```

sets `ot` pointing to `C_otbl`. Then

```
ot->base[1].offset // has the value 16.
```

Adding `16` to `pC` sets `pB` pointing to the object representing the `B` part of `C`, and the `optr` for that object points to `BinC_otbl`.

The cast to `V*` is implemented as follows

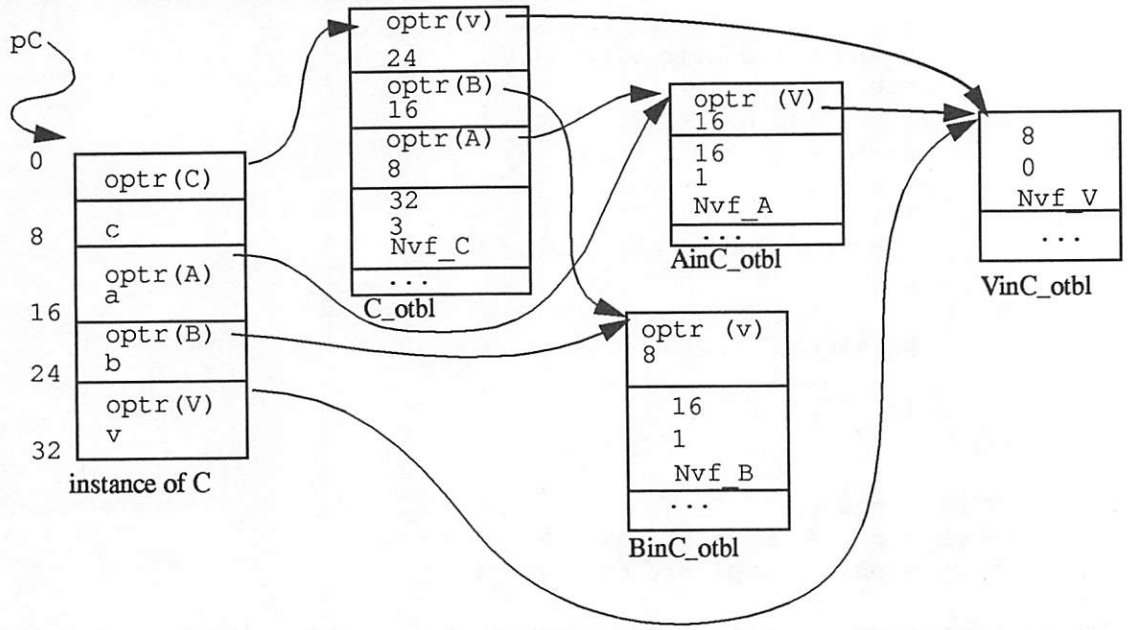
```
ot = pB->optr;
```

sets `ot` pointing to `BinC_otbl`.

```
ot->base[0].offset // has the value 8.
```

Finally, adding `8` to `pB` sets `pV` pointing to the object representing the `V` part of `B` (and of `A` and `C` since it's virtual), and the `optr` for that object points to `VinC_otbl`.

FIGURE 7. Layout of derived class instance and otbls



5.7 Composition

By *composition* we mean the declaration of a member of a class which is itself of a class type. This is the common use o embedding one class in another as in:

1. The GNU C++ compiler, `g++`, uses a similar scheme to implement virtual derivation in general where the virtual base pointers are replaced by offsets stored in the `vtbl`.

```
extern "shared" class A { public:int a1, a2;};
class B {
public: int b1;
        A ba;
        int b2;
} myB;
        myB.b2 = 999;
```

The offset of the member `b2` of `B` is dependent on the size of `class A`, which must be looked up in the `otbl`. So the code to make the assignment to `myB.b2` looks like

```
* (int*)((char*)&myB + OFFSETOF(B::ba) + myB.optr->size)) = 999;
```

5.8 Static Data Members

Static data members are unaffected by a "`shared`" linkage specification.

5.9 Static Class Functions and friend functions

For static class functions and friend functions, there is no access to the private data members. This is because these functions do not vector through the virtual function table. There is thus no way to ensure that the compile-time correspondence of the private data of the `this` pointer actually corresponds to the version of the implementation that is reached by these functions.

5.10 Constructors, operator new and sizeof

As in default linkage `operator new` is always called before the constructor. Note that allocating an object of a class with shared linkage always requires a call to `sizeof` no matter what the storage class the user specifies. Moreover in shared linkage `sizeof` is not a compile-time operator, but requires a lookup of the `otbl`. We can illustrate its implementation with pseudo-code as follows

```
size_t sizeof(TYPE) { return TYPE_otbl.size; }
```

But since there might be more than one implementation of type, it is more interesting to use `sizeof` on an instance.

```
size_t sizeof(void* inst)
{
        void* derived = ((char *)inst) + inst->_optr->most_widen_offset;
        __otbl* ot_of_derived = *((__otbl**) &derived);
        return ot_of_derived->size;
}
```

5.11 Initialization of Otbls

Since values in the `otbl` representing instance size, base object offsets, and virtual functions are not known until run-time (when the dynamic linker is run at program initialization time, or after a `dlopen`) the `otbl` cannot be initialized statically. However it must be initialized before any instance of the class can be allocated. In particular, the `otbl` must be allocated before any call to `sizeof`. Thus the `otbl` can be initialized by `.init` code in the shared object, or by code run dynamically the first time `sizeof` is called for the type and the result cached for further use.

The initialization of the `otbl` is done by calling a built-in static member function for each class with shared linkage, `otbl_creator_function`. Figure 9 below describes the otbl creation algorithm.

FIGURE 8. Algorithm for the initialization of otbls

```
__otbl* MY_TYPE::otbl_creator_function()
{
__otbl *my_otbl ← allocate(sizeof(fixed_part) +
            n_bases * sizeof_(base_part) +
            n_functions * sizeof(function_part));
my_otbl->n_functions ← MY_TYPE_NUMBER_OF_FUNCTIONS;
my_otbl->n_bases ← MY_TYPE_NUMBER_OF_BASES;
my_otbl->size ← sizeof(MY_DATA) + sizeof(_otbl*);
my_otbl->most_widen_offset = 0;

//Size and assign position of all base classes
foreach direct base otbl_b of MY_TYPE in traversal order do
   {
   my_otbl->base_b ← call TYPE_b::otbl_creator_function();
   // unify previous allocated virtual base class
   foreach base otbl_bb of otbl_b
      {
      is otbl_bb already allocated virtually?
         {
            delete base_bb;
            base_bb ← previously allocated otbl;
         }
      else // Not seen yet. Allocate its region.
         {
            base_bb->most_widen_offset = - my_otbl->size;
            my_otbl->size += base_bb->size ;
         }
      }
   }
// Assign the values for the virtual function part
foreach base otbl_bbb both direct and indirect
   {
   foreach function_f in base_bbb
      {
      if MY_TYPE overrides function_f
         {
         base_bbb->function_f->fct ← function;
         base_bbb->function_f->this_adj ← 0;
         }
      else
         base_bbb->function_f->this_adj ← base_bbb.widen_offset;
      if function_f has a covariant return type == MY_TYPE
            base_bbb->function_f->ret_adj ← COVARIANT_TYPE_OFFSET_f ;
      }
   }
return my_otbl;
}
```

5.12 Virtual Function Calls

Calling a virtual function for a class with shared linkage involves an indirection through one or two `otbls`, depending on whether the function is defined in the derived class or inherited from a base class. Figure 9 below illustrates the implementation.

FIGURE 9. Virtual function call in shared linkage.

```
class A {
public:
      virtual void vfA1();
      virtual void vfA2();
};

class B : public A {
public:
      void vfA1();        // override
      virtual void vfB();
};

B* pB = new B;

pB->vfB();                // (1)
pB->vfA1();               // (2)
pB->vfA2();               // (3)
```

This is implemented approximately as follows. Note that we show the implicit `this` parameter for purposes of illustration only.

```
__otbl* ot = pB->optr;

function_part* fp - &(ot->function[1]);                    // (1)
fp->*fct(this + fp->this_adj);

fp = &(ot->function[0]);                                   // (2)
fp->*fct(this + fp->this_adj);

ot = ot->base[0].base;                                     // (3)
fp = &(ot->function[1]);
fp->*fct(this + fp->this_adj);
```

Note that in the third case the client code references the *nested otbl* for class A in class B (nested tables are built even in the case of single inheritance). Thus in later versions derived classes can add overrides of base class virtual functions.

5.13 Overriding the return type of virtual functions

To implement calls of virtuals in which the return type has been overridden we use the same mechanism as for ordinary virtual function calls with the additional step of adding the return adjustment stored in `ret_adj`. Note that in fact *both* adjustments must always be done, since at run-time there is no knowledge of whether or not the function return type has been overridden.

This implementation does not support cases where the overriding contraviant return type is a class with shared linkage or is virtually derived from the original type. A full implementation requires the execution of a complex code sequence that calculates the return adjustment based on the actual type of the object.

5.14 Pointer to Member-Function

The pointer-to-member-function machismo is similar to the mechanism described in Section 5.5 on page 10. Pointers to member functions are represented by a structure `__smfptr`, defined as follows

```
struct __smfptr {
        size_t base_index;      // index of base
                                // -1 if direct
        size_t index;           // index of virtual
                                // negative if non-virtual
        fptr  faddr;            // only for non-virtual
};
```

Note that the offset of the `optr` need not be stored, since it is always at offset zero. The base index is used to calculate the adjustment to be applied to the `this` pointer, by a lookup in the `otbl`.

6 Instruction Counts

Most operations are identical to cfront. This includes getting data and calling virtual functions at the level of the type the data and function is defined at. The only differences occur when accessing data or calling a function in a base class. For the cfront implementation model, the are different code sequences for virtual and non-virtual inheritance. For the OBI implementation model, the code sequences are the same for virtual and non-virtual inheritance. Table 5 summarizes the instruction count difference.

6.1 Calling a virtual function defined in a base class.

This is case for current cfront non-virtual inheritance: 4 instructions

```
ld    [%i0+8],%l1              ! load the vptr into l1
ldd   [%l1+8],%l2,%l3          ! load double the thisDelta
                               ! and function addr into l2,l3
add   %l2,%o0,%o0              ! adjust the thisDelta
call  %l3
```

This is the case for current cfront virtual inheritance from a virtual base: 4 instructions

```
ld    [%i0+OFFSET_PB],%o0      ! load pB->PB into o0
ld    [%o0+8],%l1              ! load the vptr into l1
ld    [%i1+12],%l2             ! load the function addr into l2
                               ! the adjusted this pointer is
                               ! already in o0)
call  %l2
```

This is the case for OBI both virtual and non-virtual inheritance call up level: 5 instructions

```
ld    [%i0],%l1                ! load the optr into l1
ld    [%l1+A_IN_B_OFFSET],%l3  ! load B's A base into l3
ldd   [%l3+8],%l4,%l5          ! load double the thisDelta and
                               ! function addr into l4,l5
add   %i0,%l5,%o0              ! add the thisDelta to this
call  %l4
```

6.2 Accessing a member in a base class

Current cfront non virtual inheritance: 1 instructions

```
ld    [%i0+OFFSET_OF_MEMBER],%l1
```

This is the current cfront virtual inheritance: 2 instructions

```
ld    [%i0+OFFSET_PB],%l0          ! load B's A base into %l0
ld    [%l0+OFFSET_OF_MEMBER],%l1   ! indirect through PA
```

This is the OBI up level for both virtual and non-virtual inheritance: 4 instructions

```
ld    [%i0], %l1          ! get optr
ld    [%l1+A_IN_B_OFFSET],%l2 ! get the base class offset
add   %i0,%l2,%l4         ! add the base class offset to this
ld    [%l4],%l5           ! return the value
```

Table 5. Summary of Instruction Counts

	call up level non-virtual inheritance	call up level virtual inheritance	get an up level value non-virtual inheritance	get an up level value virtual inheritance
cfront	4	4	1	2
OBI	5	5	4	4

We believe that the overhead indicated is well within our acceptable levels. On modern multi-stage pipelined architecture, we believe that the actual execution time will be insignificant. Further study of the dynamic properties of the implementations are underway.

7 Object Version Migration

So long as the interface changes which resulted in the new version of a class are upward compatible, client code which was compiled against the old version works correctly when passed a "new" object. Although we recommend against exposing data members in software interfaces, it is still possible to have public and protected data members in this scheme. Public and protected data members are still present and are located at fixed offsets within the defining class. Our upward compatibility rules require that no public or protected members are reordered or deleted. Virtual functions retain the same index within the otbl which defines them (since upward compatible changes do not reorder or delete virtual functions).

A problem however arises when code compiled against the new version of the interface gets hold of an "old" object. How can this happen? This can arise if for example we have the following sequence of versions of an interface.

```
// version V1.0
class Bar { ... };
void Bar::f( Bar *pb ) { ... }

// version V1.1
class Bar { ... };
void Bar::f( Bar *pb ) { ... pb->g(); ... }
void Bar::g() { ... }          // new in version 1.1
```

If libraries implementing both versions of class `Bar`, V1.0 and V1.1, are linked into the program's address space, then "new" code (in V1.1) can be passed an "old" object (created by V1.0) and attempt to invoke an operation - such as `g()` - on it which the V1.0 object does not support.

At the present time we are exploring possible solutions to the version migration problem. Note that in the case where shared linkage is used only to support evolution, and only one version of a library is linked to a program at one time, there are no such "version migration" problems. Among the approaches we have considered are:

(i) Perform a runtime bounds check on the virtual function invocation, testing the `otbl` index against the value of `n_functions`. A similar check is necessary for access to public and protected data members.

Disadvantages of this approach are that it must be performed for *all* calls, and that it is difficult to recover after a failed check (possible actions are to call `abort` or raise an exception.)

(ii) "Migrate" the old `otbls` to new `otbls`.

In this case the entries for unsupported virtual functions are set to point to runtime error routines. This ultimately has the same error semantics as in option (i) , but there is no overhead of bounds checking for virtual functions. (In this way it is analogous to the use of thunks in a conventional `vtbl` to avoid unnecessary adjustments to the `this` pointer.)

(iii) Separate the notions of *compile-time type* and *run-time type*, and use the run-time `typeid` operator to give the programmer explicit control.

We observe that we wish the compile-time type system to treat all conformant versions of a type as the same (e.g. allow any version of the type `Bar` to be passed to a function declared as taking a parameter of type `Bar`), but that we may wish to distinguish between them at run-time. In other words applying the `typeid` operator to objects of different versions of a type would produce different results. The programmer could then explicitly test for version conformance and make appropriate decisions.

(iv) Reify the notion of namespace and allow it to dynamically correspond to a physical entity such as a dynamically linked shared library. For example create a construction which allows a namespace to be returned from a system level operation. `For example: namespace foo = dlopen(..);` Use the normal C++ type system to enforce conformance

At the present time we do not consider any of the approaches listed above to be satisfactory and we remain interested in exploring further approaches. Although (iv) is the most intriguing, we do not believe the C++ language can tolerate any additional major extensions.

The semantics we really require are those of the "newer is better" variety. A link time mechanism, such as that in SunOS 4.x, which supports a (major, minor) version pair to denote compatible versions provides the correct semantics. A change in major version denotes an incompatible change. A change in minor version denotes a compatible change, and all compatible changes are serialized i.e. there is no branching of compatible changes. The binary file resulting from the compilation and link-editing of a library or application records the version number of the dynamically linked library on which it depends. At run-time the dynamic linker selects according to its search rules a version of the library which is the "newest", which matches exactly in major version number, and for which the minor version number is greater than or equal to the minor version number specified in the client's dependency list. Thus there is only one version of a type present in the client's address space and it supports at least the functionality required by the client.

8 Conclusion

We have presented a new object model called the *Shared Object Model* and a new C++ implementation model called the *Object Binary Interface* (OBI. These models restrict some semantics of C++ programs to allow objects to span versions and to allow multiple implementations of an interface to co-exist within an address space. The shared object model supports virtual functions, public and protected data as well as derivation. The principal restriction over the normal C++ object model is that private data may only be accessed either explicitly or implicitly through the "this pointer." By using the linkage specification of C++ and allowing this key restriction of C++ usage, we are able to add an evolutionary version object model without the need for a language extension. Initial performance estimates indicate that this strategy is viable and adds little overhead to programs.

9 References

[Bancilhon] F. Bancilhon, C Delobel, P. Kanellakis, *Building an Object Oriented Database System - The Story of O_2*. Morgan Kaufman, San Mateo CA, 1992.

[Bannerjee] J. Banerjee, W. Kim, H. Kim, H. Korth. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", Proceedings of the ACM SIGMOD Conference, pages 311-322, 1987.

[Birrell & Nelson] A. D. Birrell & B. J. Nelson, "Implementing Remote Procedure Calls," ACM Trans. Computer Systems, 2(1), February 1984

[Ellis & Stroustrup] M. Ellis & B. Stroustrup, *The Annotated C++ Reference Manual*, AW 1990.

[Goldstein] T. Goldstein, A. Sloane, M.Ball, A. Palay, "Supporting the Evolution of Class Definitions - Workshop Report" in OOPSLA'93 - Addendum to the Proceedings, to appear in SIGPLAN Notices, 1994.

[Hamilton & Radia] G. Hamilton and S. Radia, "Using Interface Inheritance to Address Problems in System Software Evolution", Proceedings of the ACM Workshop on Interface Definition Languages, 1994.

[Harrison & Ossher] W. Harrison and H. Ossher "Extension by addition: Building extensible software." Research Report RC16127, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 1990

[Lenkov] D. Lenkov, M. Mehta, S. Unni, "Type Identification in C++", Proceedings of USENIX C++ Conference, Washington D.C. April 1991.

[OMG] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, August 1991, OMG Document Number 91.8.1.

[Palay] A. Palay, "C++ in a Changing Environment", in Proceedings of the Usenix C++ Technical Conference, Portland, September 1992, pages 195-206.

[Penney & Stein] D. Penney & J. Stein, "Class modification in the GemStone object-oriented DBMS", in Proceedings of the ACM OOPSLA Conference, pages 111-117, 1987.

[Richardson and Carey] J.E. Richardson and M.J. Carey, "Persistence in the E Language: issues and implementation," *Software—Practice and Experience* number 19, pages 1115-1150, 1989

[SOM] *OS/2 2.0 Technical Library System Object Model Guide and Reference*, IBM Doc 10G6309.

[Stroustrup] B. Stroustrup, "Namespaces", ANSI X3J16/93-0105, ISO WG21/N0312.

10 Acknowledgments

The authors gratefully acknowledge the help provided by Michael L. Powell of SunSoft and Michael S. Ball of SunPro, as well as all of the reviewers on the program committee. We would especially like to thank Brian T. Lewis for the thorough careful reading and editing of this paper.

A Framework for Building Extensible C++ Class Libraries

Arindam Banerji, Dinesh Kulkarni, David Cohn
Distributed Computing Research Laboratory
University of Notre Dame
Notre Dame, IN 46556
axb@cse.nd.edu

Abstract

Extensibility leads to better designed and more reusable software. Traditionally, implementors have built extensible C++ software using ad hoc mechanisms built from scratch. This paper identifies specific characteristics that constitute extensible software. A framework for building extensible C++ libraries has been defined and constructed on AIX 3.2. Finally, the paper gives guidelines for implementors of extensible software through a discussion of an on-going application of the framework.

1. Introduction

A critical problem in designing software libraries is the difficulty of predicting possible future uses. Designers of C++ class libraries attempt to avoid this problem by *designing in* implementation choices. However, basing libraries on most probable uses and using run-time checks to select an implementation cannot accommodate unforseen future uses. Nowhere is this problem more acute than in the construction of system services to handle mobility or workstation clustering. Three examples are presented below.

Consider a library for managing information access in a mobile computing environment [Banerji, 93a]. The caching behavior and access mechanisms are critical to the performance of such a system. Optimum caching performance is highly dependant on usage patterns which can neither be accurately predicted at design time nor reasonably calculated at run-time. Rather, performance will improve if the implementation can respond to client-provided run-time control directives. Run-time client control over the implementation, or just the ability of a client to provide usage hints, could better optimize performance.

In a clustered workstation environment [Banerji, 93b], on-line updating of software is desirable. Ideally, a new implementation could be added to running software without disturbing existing applications. This implies the ability of a client to dynamically replace or add implementations of either member functions or even entire classes.

Finally, suppose that a large company has developed a class library to manage its critical corporate data. The library includes mechanisms to handle all inquires in use at the time of its creation. Later, after the developers have moved on to other projects, a new need arises. If this need cannot be handled by the existing library, either the developers must return to this project or new programmers must wade through the original source code. If the library were extensible, new functionality could be added without reference to the original code or its developers.

Although the situations presented above may seem far fetched, they are real problems in

mobile computing, clustered computing and management of object library management [Banerji, 94]. Moreover, these problems have counterparts in system and application software for standalone workstations [Krueger, 93]. As discussed below, some solutions have been suggested in the past. However, many of these solutions are ad hoc and lacked the genericity needed to construct extensible class libraries for other application domains. This paper addresses genericity by discussing the design, implementation and use of a *framework* that allows C++ class library designers to build extensibility into their subsystems.

This framework has various specializable and easily replaceable components. In addition, certain programming guidelines that aid in gluing together the components of an extensible library are discussed. The existing libraries for this framework and sample test cases have been built for AIX 3.2 running on IBM RS/6000s. A custom port of the AT&T cfront 3.01 was used to build the components.

The next section discusses extensibility and its implications. The terms and technologies pertaining to the framework implementation are then detailed. Related work is discussed in the subsequent section. Section 4 lays out the overall structure of the framework as well as the details of the design principles. Section 5 discusses the programming guidelines that implementors need to follow and presents a concrete example. Section 6 describes the user's view of software, built using this technology. The paper ends with a summary of its contributions.

2. What Extensibility really means

This section attempts to demystify the term *extensibility*. In order to do this, it is necessary to list the features that usually characterize extensible software. The identification of such features leads to a better understanding of the techniques and mechanisms necessary to support extensibility.

Extensibility implies different things to different people. Instead of using some preconceived notion of this software property, we choose to define the essential features of extensible software. Some of these features are obvious, while others are not. The features are:

- ***Separation of interface from implementation*** is necessary to ensure that the visible functionality of a software library is not cluttered with non-relevant implementation details. Such a feature is considered good practice, since it allows for implementation changes with little or no client-code recompilation. Furthermore, it is a fundamental requirement for supporting other features of extensible software.
- ***Tunable implementations*** allow design decisions to be based upon actual run-time data. Best case scenario implementations are replaced by designs that can act upon client-provided hints and directives. Thus, application-dependant usage information, which can never be predicted at design time, may be used by clients to fine-tune software implementations. This feature may appear to conflict with interface-implementation separation. However, as discussed later, it is possible to simultaneously support such conflicting features.
- ***Multiple coexisting implementations*** for one particular interface allow clients to choose an implementation that closely matches their needs. Hence, conflicting design choices may be reflected in separate implementations, thus affording implementors the luxury of application-specific optimizations. Clients on the other hand, are left with the responsibility of selecting an appropriate implementation at run-time. Furthermore, this separation allows for the independent development of different implementations of an interface. Thus, bug fixes for exist-

ing implementations can be made available as new implementations of an existing interface.

- ***Addition or substitution of implementations*** allows clients to effect major reconfigurations without recompilation or even application restart. Constituent classes and member functions of an implementation may be dynamically replaced. Similarly, whole new implementations for existing interfaces could be loaded at run-time. Typically, this is used to dynamically load new implementations of existing interfaces, in order to fix bugs or update services.
- ***Dynamic addition of member functions to interfaces*** allows clients to add services without requiring recompilation of existing code. Usually implementations are expected to support only those public member functions that correspond to a particular interface. Should a new implementation support additional member functions, the clients can effectively integrate such services by extending the interface dynamically. Such integration does not affect existing clients who still can use the old unextended interface.

These features of extensible software point to specific base technologies that are required to implement them. These base technologies have been adopted by the flexibility framework, described in this paper. These technologies, in no particular order, are:

- ***Explicitly separate interface and implementation hierarchies:*** Most well designed software systems separate out interfaces from implementations. Typically, such separations are performed in an adhoc manner and the interaction between interfaces and implementations is decided on a case-by-case basis. In order to support separate implementations, it is necessary to completely separate out interfaces and implementation hierarchies. The inter-dependence of these hierarchies, may be minimized by ensuring that interface classes only interact with an abstract base class representing the implementation hierarchy. All concrete implementations of an interface inherit from this base class. However, the interface classes do not depend upon the symbols of any specific implementation; only on those of the abstract base class.
- ***Run-time access to type information:*** Interactions between interface and implementation hierarchies involve passing pointers to abstract base class of the implementation hierarchy. This is necessary to ensure that the interface classes do not depend upon any symbols available in concrete implementations. Thus, downcasting to derived class pointers is often required. Classes that are used in the interaction between interfaces and implementations are thus, associated with Run-Time Type Information [Lajoie, 93].
- ***Dynamic linking and loading:*** Run-time addition of implementation requires that object modules be loaded and linked into running code. Similar facilities are necessary to enable on-the-fly substitution of implementation classes. Whatever mechanism is used has to deal with C++ specific problems, such as, mangled names and initialization of static constructors and destructors. Invariably, such a mechanism is highly dependant upon the exact nature of the dynamic linking services provided by the target operating system.
- ***Class objects:*** Flexibility to control implementations or manipulate the member functions that belong to an interface requires a level of indirection greater than that provided by C++. Class objects are used to provide this indirection. As in Smalltalk [Goldberg, 89], class objects also afford a mechanism for calling C++ constructors, based upon a given set of properties, instead of the name of the class being initialized. These objects may be associated with interface and implementation objects to act as a run-time interpreter of some usually implicit entity.
- ***Indirection in name resolution:*** Dynamic addition of member functions to an existing class interface requires that it be possible to map a given member-function call to a generic

per-class function-call dispatcher [Coplien, 91]. This technology of indirection in name resolution or dynamic dispatch, already exists in many pure object-oriented languages [Ungar, 87]. However, making it available for C++ with reasonably good performance, is quite another matter.

- ***Dual interfaces:*** If implementation details are removed from interfaces, how can clients of extensible software fine-tune implementations? The technology of opening up implementations through dual interfaces is used. The idea of open implementations [Kiczales, 92] with one interface to access the functionality of a subsystem and another to optionally control the implementation, is not new. It allows clients to provide usage information and implementation directives through a second interface. The second interface or the meta interface should be made available to clients on a need-to-know basis.

All these base technologies have existed for a while. However, we believe no one has integrated them to provide an environment explicitly geared to developing extensible C++ libraries. This work integrates these existing technologies into an easily usable form.

3. Related Work

The need for extensibility in software has been stressed for both operating systems and languages [Kiczales, 92]. The authors of the Meta-object protocol for CLOS [Kiczales, 91] have been instrumental in discussing open implementations and dual interfaces. Apertos (formerly Muse) [Yokote, 92] has applied reflection and meta-object protocols to operating systems. Choices [Campbell, 93], an object oriented operating system has done yeoman work in supporting frameworks for dynamic code loading and stepwise refinement. Recently, Open C++ [Chiba, 93] has used translator directives for redirecting method invocations to metaobjects to implement object groups in a distributed system.

Many of the basic mechanisms, such as changing type interfaces on the fly, have also been discussed in previous work. These include preprocessor-generated class objects to instantiate dynamically loaded subclasses [Dorward, 90] and type-set interfaces for schema manipulation in databases [Skarra, 86]. Of course, the RTTI extensions of ANSI C++ have been discussed in great detail [Stroustrup, 92], [Vines, 93]. Finally, it is important to note the numerous idioms mentioned in the last few chapters of [Coplien, 91] that deal with flexibility, indirection and its effective use.

Finally, it is pertinent to mention some recent industry initiatives. Although, efforts such as CORBA [OMG, 91] effectively separate out implementations from interfaces, they are mainly geared towards facilitating object interactions. Thus, objects created by two different languages can be made to cooperate as in IBM SOM [IBM, 91] or object interactions can be transparently made to span machine boundaries. It is possible to associate some flexible properties to CORBA compliant software through the use of meta-objects and indirections. However, very little support is provided for structuring object systems for flexibility.

4. The Extensibility Framework

This framework represents a significant set of collaborating classes and class hierarchies, that capture the patterns and mechanisms needed to implement extensible C++ software[1]. Each of the components or class-hierarchies of this framework can be further specialized by using inheritance.

1. Paraphrased from [Firesmith, 1993]

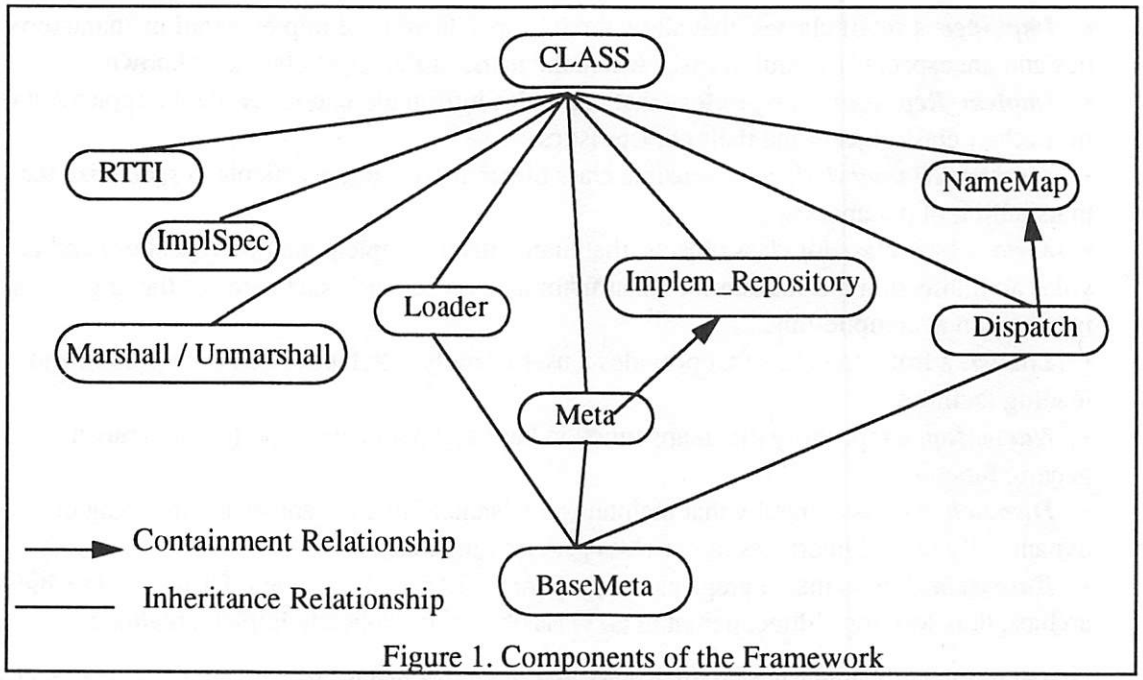


Figure 1. Components of the Framework

This allows programmers to tailor the services provided by this framework according to their specific needs. In addition to these components, there are certain programming guidelines both for implementors who use this framework and for clients[1] who use the software built with this framework. This section discusses the details of the various components of the framework shown in Figure 1. The next section presents how such a framework may actually be used.

Before reviewing the components of the framework, it is important to understand the three distinct relationships between the entities discussed here. The first one is the *instance-of* relationship between a class and its object instance. The former is represented only in the C++ source while the latter also has a run-time representation. The second one is *inheritance* which applies to both interfaces (subtyping) as well as objects (implementation). The third relationship is more subtle: that between an object and its metaobject. The latter provides a run-time representation of some of the implementation aspects of the former. Since a metaobject makes an otherwise implicit aspect explicit, it is said to *reify* that aspect. For simplicity, a metaobject that reifies a class will be referred to as a *class object.*

Figure 1 shows the various component hierarchies of the extensibility framework. There are three kinds of components - those that are incorporated into class-objects, those that are part of every object and those that realize some helper services. Typically, as mentioned below, the BaseMeta component is part of every class object. On the other hand, RTTI is part of every object. The Marshall/Unmarshall component and the ImplSpec component facilitate interactions between interfaces and implementations. The components in Figure 1 are:

- ***CLASS***, a base class from which every other framework class inherits, is a place-holder which ensures the safety of type-casts.
- ***RTTI***, the set of Type_info and typeid classes that implement ANSI compliant run-time type information for appropriately instrumented classes.

1. Henceforth, "implementor" refers to programmers who use the framework directly to build software and "client" refers to those who use the software developed in this manner.

- ***ImplSpec***, a set of classes, that allow flexible specification of implementation characteristics and are especially useful in cases where the name of the target class is unknown.
- ***Implem_Repository***, a repository that maintains information about available implementations, their class objects and their chracteristics.
- ***Marshall/Unmarshall***, an extensible class hierarchy that allows clients to specialize the marshalling of parameters.
- ***Meta***, a base class for class objects, that maintains the implementation repository and provides an indirection mechanism for constructor calls, when the exact name of the target class is unknown at compile-time.
- ***Loader***, a front-end class that provides a user-friendly interface to run-time linking and loading facilities.
- ***NameMap***, a repository that maps function keys and parameter type-tags to a target generic function.
- ***Dispatch***, a class hierarchy that maintains the NameMap and handles the mapping of dynamically loaded interfaces to per-class generic functions.
- ***BaseMeta***, a class that aggregates the properties of Meta, Loader and Dispatch class hierarchies, thus forming a direct parent to all class objects for concrete implementations.

Each of these components is actually a class hierarchy that can be specialized to get the precise functionality desired. Implementors incorporate elements of these class hierarchies to effect extensible software. Although, implementors and clients only see these components, the actual realization of the framework uses a few libraries to effectively implement some services. There are primarily two such libraries - the run-time linking library for AIX and the library that implements RTTI. These libraries acts as implementations tools for this framework. The next few subsections discuss some of the basic elements shown in the figure. The section ends with a brief description of the implementation tools or libraries used by this framework.

4.1. Identification of Different Implementations

As mentioned before, a particular interface may be supported by any number of different implementations. Hence there must be a mechanism to identify these different implementations. Since, it is possible that implementations may be loaded at run-time, names of implementation classes cannot be used for such a purpose. Instead, the `ImplSpec` hierarchy provides the mechanism necessary to identify some particular characteristics of an implementation. `ImplSpec`, as shown below, defines a common protocol for comparison of implementation characteristics. Subclasses of the class `ImplSpec`, may use any characteristic such as the implementation-name to identify implementations, but have to support the comparison protocol defined by `ImplSpec`. Thus, typically every piece of extensible software defines its own subclass of `ImplSpec` in order to distinguish between different implementations.

```
// The following class defines the common comparison protocol, to be supp
// ported by all characterizations of implementations. Different exten
// sible software libraries may characterize implementations differ
// ently.Some may do it through implementation-name strings, while oth
// ers may use integer constants.

class ImplSpec : virtual public CLASS // CLASS is a global base, associ
// ated with properties, common to all objects of an extensible library.
      {
       public : // some operations have been omitted for brevity
```

```
        ImplSpec(const char *); // all characterizations are finally
                                // converted into strings, for simplicity.
        virtual ~ImplSpec() ; // destructor
        // The comparison protocol follows
        virtual int operator == (ImplSpec &) ;
        virtual int operator == (ImplSpec *) ;
        virtual int operator != (ImplSpec &) ;
        virtual int operator != (ImplSpec &) ;
        ... other comparison operators ...
    private :   // mainly responsible for maintaining pointer to a
                // global repository, which catalogs all available
                // implementation characteristics.
        impl_map_t *table; // implementation repository front-end
        ...other private data...
    } ; // Base class of the Implementation Specification hierarchy
```

4.2. Indirection as an Architectural Tool

The key to building extensibility is indirection. For example, in C++ virtual functions provide a level of indirection that allows a call of the form `base_object_ptr->foo()` to be dynamically resolved to the function `foo`, in an appropriate derived class object. However the semantics of C++ limits the target of the resolution to be a similarly named function within the class hierarchy. If static type-safety were not a concern, another level of indirection could have removed the shackles of a fixed resolution mechanism. Thus, it is very important to figure out exactly where an indirection should be added and what may be gained from adding it. Based on this, the framework adds the following indirections:

- Indirection in constructor calling
- Indirection in name-resolution during function-dispatch
- Indirection in marshalling/unmarshalling parameters

4.3. Constructor Calling

Constructors for dynamically loaded classes need to be called indirectly, since class-name-based constructor calls would cause unresolved externals during compilation. The class `Meta` and `Implem_Repository` cooperate to provide this indirection. All classes which need this degree of flexibility are associated with a class object, that is a specialization of `Meta`. `Meta` in turn contains a pointer to a globally available instance of `Implem_repository`, as in

```
class Meta: public CLASS {
            public: // ignore constructors and destructors
/*
In the following code ImplSpec is used to specify some characteristic of
the implementation. In the simplest case, it may simply be a string con-
taining the name of a class.
*/
                virtual void register_class_object(ImplSpec *,...);
                 // function that allows addition of the class-object
                 // of a subclass[1] to the repository.
                virtual void unregister_class_object(ImplSpec *,...);
```

1. Subclass here refers to the subclass of the class that the class that Meta is associated with. This, if `foo` is associated with `fooMeta`, a specialization of `Meta`, then the subclass here refers to `childof_foo`, a class derived from `foo`.

```
        // removes the entry for class-object of a sub-class
        // from the repository.
      virtual CLASS *instantiate(ImplSpec *);
        // an indirect constructor called with some form of a
        // of a specification of which particular subclass
        // needs to be instantiated.
private:
      static Implem_Repository *repository;
};    // specification of the Meta class.
```

Typically, dynamically loaded code is in the form of a subclass of an existing base class. Assuming that the base class and the subclass are associated with class-objects derived from Meta, then the sub-class automatically registers its class-object with the super-class repository. Thus, if `foo` and `fooMeta` are respectively the superclass and its associated class object and `childof_foo` and `childof_fooMeta` are the derived class and its associated class object, then the following happens during static initialization:

```
childof_fooMeta childof_foo::meta = new childof_fooMeta;
```

This call to the constructor of `childof_fooMeta` causes it to register itself with `fooMeta`, as the class object for `childof_foo`. At this point, instantiation requests for `childof_foo` when sent to `fooMeta`, are automatically forwarded to the `instantiate` in `childof_fooMeta`. This `instantiate` function, in turn calls the constructor of `childof_foo`. If there are additional parameters that need to be passed to the constructor of `childof_foo`, then the code gets a more complicated, but the principle remains the same.

4.4. Name Resolution and Function Dispatch

Quite often, a subclass with an additional non-inherited member function, needs to be dynamically loaded. This implies that the supported interface is extended by the addition of this subclass and clients using this new subclass should be able to access this new member function. Since, in C++, a named function call always gets resolved to a similarly named function, an extra level of indirection is needed. In this case, the name resolution mechanism during function-dispatch needs to be extended. This indirection is provided by the classes `Dispatch` and `NameMap`. These classes are quite similar to the two classes discussed above. Instead of the function `instantiate`, the function `generic_func` of the following form is used.

```
CLASS *generic_func(int function_key, CLASS *,...);
```

The dispatch object of the loaded subclass, in a manner similar to the one shown above, registers itself with the `NameMap` of the superclass dispatch object. This ensures that when a client directs an extended call to the newly loaded subclass, the generic function of the dispatch object of the subclass is ultimately called. The generic function then calls the appropriate function in the subclass.

4.5. Marshalling and Unmarshalling

For distributed systems, which form a large part of our research focus, the client and the implementation of the member function may be separated by machine boundaries.This typically requires marshalling and unmarshalling of parameters. However, factors such as relative alignments of the target and source architectures and the kind of communication links available may have a tremendous impact on the way marshalling and unmarshalling is done. In order to ensure

that best possible performance is guaranteed, a slightly different form of the generic function is used.

```
Marshalled_Parm *generic_func(int, Marshalled_Parm *);
```

In this case, the class `Marshalled_parm` is an implementor specialized class that allows custom parameter marshalling and unmarshalling schemes[1]. These architectural components are aided in great part by two major implementation tools that are discussed below.

4.6. Implementation Tools

The dynamic linking and loading tool consists of a code loading library and a `Loader` class. The code-loading library, dynamically links in and loads C++ libraries into running programs. At present, it only supports the XCOFF file format of AIX 3.2. The interface supported by the library mimics the SUNOS run-time linker calls of `dlopen, dlsym and dlclose`. In order to be dynamically linkable, the set of object modules pertaining to the loaded subclass are archived into a custom shared library. Initially, the appropriate object modules of the subclass and its class objects are linked into a single object module with all outstanding externals unresolved. This object module is passed through `Munch` [USL, 92], to create a list of static constructors and destructors. A set of entry-points are created to enable calling these destructors and constructors[2].- Finally, a shared library that archives the single object module, the static initializers and the generated entry-points, is created. As is obvious, the compiler driver of AT&T's `cfront` had to be changed somewhat to support this process.

During the call to `dlopen`, all unresolved externals within this library are bound to the appropriate symbols of the running client program. The `dlopen` and `dlclose` calls also ensure that the entry-points for calling static constructors and destructors, get called automatically.

The `Loader` class, in turn provides a simplistic interface that hides some of the complexity of dlopen. Furthermore, this class implements an automatic lookup of certain directories to locate the requested loadable library. The main interface function of this class is:

```
int add_impl(char *BaseClassName, char *LoadTargetCharacteristic);
```

The RTTI tool is a library-version[3] of the ANSI-C++ language extension. The Type_Info and typeid classes are closely based upon the implementation detailed by Bjarne Stousroup [Stroustroup, 91]. An extensive set of macros have been added that ease the chore of instrumenting classes. For example, the declaration of a class needs to include a line of the form:

```
RTTI_SCAFFOLDING_DECL(NAME_OF_CLASS)
```

The definition of the class needs to include a macro of the form:

```
RTTI_SCAFFOLDING_IMPL1(NAME_OF_CLASS, NAME_OF_PARENT_CLASS)
```

The macros for handling template classes are equally easy to use. A set of macros have been provided to automate narrowing of classes, in the presence of virtual inheritance. The set of RTTI classes themselves may be easily specialized as per the requirements of the ANSI standard.

---

1. Marshalling/Unmarshalling is not discussed in detail, since it is only of interest in case of distributed computing.
2. These correspond to `_main` and `__dtors` in the USL `libC.a`, except that they have different names.
3. It is available for ftp from invaders.dcrl.nd.edu:/pub/software/rtti.tar

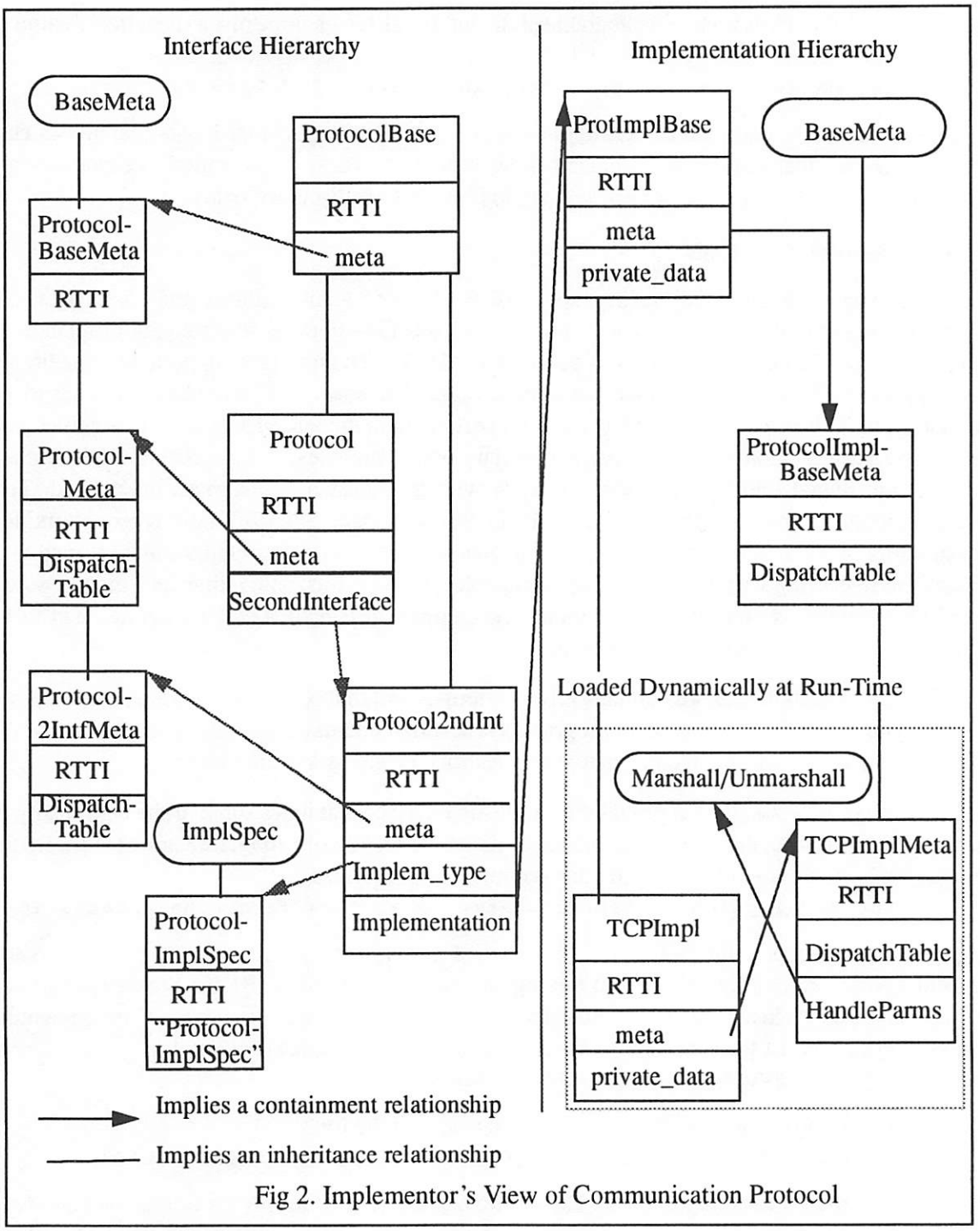


Fig 2. Implementor's View of Communication Protocol

5. Implementor's view

Having discussed the internal design of the framework components, this section sheds light on the overall structure of extensible software, that may be constructed using this framework. One of the target subsystems, currently under construction is an implementation of a user-level communication protocol library. The idea is to provide a dynamic object-oriented framework for building communication protocols, a problem similar to that addressed by x-kernel [Peterson, 90]. The layout of the hierarchies that constitute the framework is shown in Fig 2.

Figure 2, clearly shows two hierarchies - the implementation hierarchy and the interface hierarchy. On the interface side, there are two main objects - one representing the primary interface of network protocols and the other, an interface for controlling implementations. Each of these objects are associated with class-specific meta-objects that provide facilities such as function dispatch. One the implementation side, all concrete implementations inherit from a single base class. Again, the base class of the implementation as well as concrete realizations that inherit from this base class, are all associated with class-specific meta-objects. The functionality of the meta-objects on the implementation side is slightly different from those on the interface side. As can be clearly seen from the figure, only a single arrow crosses the interface-implementation barrier. This arrow represents a polymorphic pointer to the implementation tree. All interactions with implementations are exacted through this polymorphic pointer. Two final points need to be made about the figure. Although not shown, all classes in the subsystem inherit directly or indirectly from the class `CLASS`. Concrete implementations, that is subclasses of the implementation base as well as any associated meta-objects, may be attached to the implementation hierarchy at run-time through the dynamic loading services.

The main interface of the communication protocol library is provided by the class `Protocol`. It actually provides two interfaces, the first one related to the basic functionality and the second one for manipulation of the implementation, or a meta-interface which is reified by an object of class `Protocol2ndInt`.

```
class Protocol: public ProtocolBase {
        public:
                // constructors and destructors
                // open a passive session
                virtual Protocol &OpenSession(...);
                // open an active session
                virtual Protocol &OpenSessionEnable(...);
                // passes certain calls to the second interface.
                Protocol2ndInt operator->();
                        ...
                /* member functions for comm. protocols. */
                        ...
                // macro that generates RTTI scaffolding
                RTTI_SCAFFOLDING_DECL(Protocol)
                // pointer to class object that controls
                // the behavioral metacomputation...
                static ProtocolMeta *meta;
        private:
                // Pointer to the optional second interface
                Protocol2ndInt *SecondInterface;
        }; // specification for the Protocol Interface
```

As can be seen in the code fragment above, apart from providing support for the functionality provided by a regular communication protocol, the class contains pointers to two other entries - a *second interface object* and a meta-object. Typically, the member functions of `Protocol` just pass on the calls to appropriate member functions of the `Protocol2ndInt` class. The meta-objects, build upon the various components of the framework, and are discussed in the next subsection.

An object of the class `Protocol2ndInt` implements the second interface. For a communication protocol the dual interface allows users to provide such directives as window-sizes, distribu-

tion of small communication buffers vs. large communication buffers etc. These features are similar to those offered by the unstructured ioctl system call. In addition, the client gets to choose which particular implementation of the protocol he/she would like to use. This information is specified in the form a ImplSpec class, which allows various kinds of implementation options to be specified.

```
class Protocol2ndInt: public ProtocolBase[1] {
      public:
            // ignore constructors and destructors...
            void set_impl(ImplSpec *,...);
            // choose a particular implementation - such as TCP.
            void set_window_size(int);
            // set the window size.
            ...
            /* other functions to support an open implementation*/
            ...
            /* If necessary, member functions to handle forwarded
                  calls from the Protocol class. */
            static Protocol2ndIntfMeta *meta;
            // class object that allows behavioral manipulation
            RTTI_SCAFFOLDING_DECL(Protocol2ndInt);
      private:
            ProtocolImplSpec *implem_type;
            // identifies which particular implementation was
            // chosen, e.g.: TCP or UDP.
            ProtImplBase *implementation; // the implementation
      }; // specification of the dual interface.
```

As can be seen from Figure 2, the only symbols from the implementation hierarchy that are visible to the interface hierarchy, are those of the class ProtImplBase - the root of the Implementation hierarchy. ProtImplBase has two main functions. Firstly, it acts as an abstract placeholder, from which any number of prototype implementations may inherit. This ensures that dynamically loaded subclasses of ProtoImplBase are used in a type-safe manner. Secondly, ProtImplBase through its meta-object manages the extensible properties of the real prototype implementations. With the help of the Implem_repository and the BaseMeta classes, it provides the services necessary to afford run-time extensibility.The message protocol between the two hierarchies is that supported by the member functions of ProtImplBase and the meta-object of ProtImplBase i.e.: ProtImplBaseMeta.The well-specified nature of the interaction between the two hierarchies, ensures that multiple implementations can co-exist.

```
class ProtImplBase: public CLASS {
      public: // ignore constructors and destructors
            // some communication protocol pertinent functions
            void openSession(...);
            void openSessionEnable(...);
            ...
            /* some communication related member functions */
            ...
            ProtocolImplBaseMeta *operator->();
```

1. The class ProtocolBase simply encapsulates, the shared characteristics of the Protocol2ndInt and Protocol classes, and thus forms a placeholder for structuring the interface hierarchy.

```
        // returns pointer to the implementation meta-object
private:
        ProtocolImplBaseMeta *meta; // the meta-object
}; // specification of the root of the Impl. hierarchy
```

5.1. An Implementor's Manual

Having shed some light on the overall structure of software that uses the extensibility framework, it is time to specify the exact steps that a programmer must take to be implement such software. For most of the implementation steps that need to interact with the framework, template files are used to guide the implementor. Generic makefiles make the task of building these executables, even simpler. Finally, it must be mentioned that certain conventions need to be followed while naming shared libraries of interfaces and loadable implementations. These conventions, not mentioned here, allow the dynamic loading facility to easily find a shared library that matches a certain implementation characteristic. The steps themselves are:

Step 1. Create a primary interface and a secondary interface class for the software library that needs to be constructed.

Step 2. Create a base class for the implementation hierarchy, which essentially handles the sum of all the member functions represented in the two classes from Step1. The definitions of these member functions may be kept empty.

Step 3. Associate each of these classes with meta-objects or class-objects, based on simple name substitution of available template classes. The instantiate function of the meta-objects on the implementation side must be updated to match the actual constructors supported by the primary interface.

Step 4. Declare a few standard static objects that allow for automatic creation of instances of the meta-objects for both the implementation and interface objects. The declaration of these static objects is facilitated through easy-to-use macros.

Step 5. Update available generic makefiles to use the actual file names used for this particular software library.

Step 6. Use the makefiles from Step 5 to create shared libraries containing the interface hierarchy, the base of the implementation and associated meta-objects. Thus, a binary form of the interface provided by the software library is now available. At this point, an implementation must be created.

Step 7. Design and create an implementation that supports at least all the member functions provided by the base class of step 2. If no extra member functions are to be supported, jump to Step 11.

Step 8. Create an implementation-specific dispatcher, based upon available templates. This dispatcher is called by generic_func, when the new member function is called by a client.

Step 9. Create a header file that defines macros to map this new function call to a call to `generic_func`. This header file can also be set up using available template header files.

Step 10. At this point create the required meta-object for this implementation and declare the requisite static objects, as in Step 4.

Step 11. Use generic makefiles, and create an implementation-specific makefile. Ensure that naming standards for implementation libraries are followed.

Step 12. Finally, place the shared library created in Step 12, in an appropriately named

directory, following certain naming conventions.

As mentioned, the objects in the interface hierarchy, the root of the implementation hierarchy and a meta-object that controls the implementation hierarchy root are archived into a shared library. A client program can then link in the shared library corresponding to the interface, that he/she needs to use. An implementor creates a new implementation, creates a loadable shared library that inherits from the implementation base class. This loadable library, can be automatically loaded at the request of the client (as discussed in Section 6). Similarly, any number of new implementations may be created and loaded.

6. Client's View

A typical client of extensible software, thus engineered may want to use the interfaces provided for three particular purposes:

- Use the direct functionality of the interface e.g.: OpenSession
- Use the second interface to control the implementation e.g.: set_window_size
- Use the meta-functionality to load new implementations of the interface.

The following piece of code demonstrates this for the protocol class, discussed above. Initially, a client programmer allocates a protocol object and selects the implementation to be used. For example, in this case the programmer decides to use the UDP implementation of the protocol. After setting the implementation, the programmer may use the UDP implementation as desired. At this point, for some hypothetical and fictitious reason, the programmer decides to use TCP, instead of UDP. Assuming, that the TCP implementation is not pre-loaded, the programmer asks for it to be loaded through the `add_impl` call. Subsequently, the TCP implementation is selected and the TCP protocol realization is ready for use. Finally, just as a demonstration sample, the programmer chooses to change the sliding window size used by the TCP realization. This is achieved through the smart-pointer which provides access to the secondary interface. It is perhaps important to mention that the following code is meant to demonstrate the use of flexibility and not necessarily to present semantically correct use of the TCP protocol.

```
 // Allocate a protocol object - the default implementation is
 // used - there may or may not be a default implementation.
Protocol *obj = new Protocol;

// Actually set the implementation to be used to be udp
 (*obj)->set_impl("UDP");
 /*the pointer operator is used to get at the second-interface*/

// Use the functionality of the protocol object now.
 obj->OpenSession(...);
... /* some code here */...

// At this point the client decides to add the tcp implementation
// to the running program. It calls the interface provided by
// the loader class, thru the meta pointer in the Protocol class.
 obj->meta->add_impl("ProtImplBase","TCP");

// Now the object may change the implementation type
(*obj)->set_impl("TCP");

// Now, the object can be called regularly, as in..
```

```
(*obj)->set_window_size(...);
```

Occasionally, the client may want to load an implementation that extends the prescribed interface. Let us assume that the TCP implementation is being used. At this point the programmer decides to use MobileTCP, a realization of TCP that supports mobility of connections. This extra functionality is supported by an additional member function **migrateconn**. One possible mechanism would be to create an new interface and use it to access the new implementation. However, sometimes recompilation of interface classes is not an option. In such cases, the programmer may dynamically extend the interface of the class, as shown below. The steps to be taken are as follows:

```
// Assumes that the appropriate header files for MobileTCP are actu
// ally pulled in by the client programmer.
Protocol *obj = new Protocol;
// Add the new implementation first
obj->meta->add_impl("ProtImplBase", "MobileTCP");

// Add the interface to the interface hierarchy - here the name of
// the function to be added is "migrateconn"
obj->meta->add_intf("ProtImplBase", "MobileTCP", "migrateconn");
// At this point migrateconn is ready for use.
obj->migrateconn(...) ;
```

At this point, other clients may directly call the migrateconn function, as long as they are using the Mobile TCP implementation. Existing clients do not need to recompile any code. However, new clients that intend to use this new function must pull in the header files specific to this extended interface, so that a call to migrateconn, automatically gets expanded into a call to generic_func. It is expected that this kind of extensibility will not be used very often.

7. Conclusion

The framework described here, represents a critical step in structuring extensible software. In addition to identifying the specific characteristics of extensible software, it provides a good set of tools for dynamic flexibility. Its services are geared towards the construction of more reusable and better designed software. Perhaps more importantly, it is a key step towards developing class libraries that can be tailored without access to source code.

8. Availability

The extensibility framework is not yet available for general distribution. Please contact the primary author at axb@cse.nd.edu, for latest availability information.

9. References

[Banerji, 93a] A. Banerji et. al. Mobile Computing Personae, *Proc. Workshop on Workstation Operating Systems IV,* Napa, California, Oct. 93, pp. 14-20.

[Banerji, 93b] A. Banerji et. al. The Substrate Object Model and Architecture, *Proc. IWOOOS '93*, pp31-43.

[Banerji, 94] A. Banerji et. al. Design, Distribution and Management of Object-Oriented Software, *Proc. USENIX Applications Development Symposium*, to appear.

[Campbell, 93] R. Campbell et. al., Designing and Implementing Choices: An Object-Oriented System in C++, *Communications of the ACM*, 36(9), Sept, 93, pp. 117-126.

[Chiba, 93] S. Chiba & T. Masuda, Designing an Extensible Distributed Language with a Meta-Level Architecture, *ECOOP '93 - Object-Oriented Programming, LNCS 707*, Springer Verlag, pp. 482-501.

[Coplien, 91] J. Coplien, *Advanced C++ Programming*, Addison Wesley.

[Dorward, 90] S. Dorward, R. Sethi, J. Shopiro, Adding New Code to a Running Program, *USENIX C++ Conference*, pp. 279-292.

[Firesmith, 93] D. Firesmith, Frameworks: The Golden Path to Object Nirvana, *Journal of Object-Oriented Programming*, 6(6), Oct. 93, pp. 6-8.

[Goldberg, 89] A. Goldberg & D. Robson, *Smalltalk-80 The Language*, Addison Wesley, 1989.

[IBM, 91]IBM (1991) *OS/2 2.0 Technical Library, System Object Model and Reference*, Version 2.00, IBM.

[Kiczales, 91] G. Kiczales et. al., *The Art of Metaobject Protocol*, MIT Press.

[Kiczales, 92] G. Kiczales, Towards a New Model of Abstraction in the Engineering of Software, *Proc. Workshop on Reflection and Meta-level Architectures, IMSA '92*.

[Krueger, 93] K. Krueger et. al., Tools for the Development of Application-Specific Virtual Memory Management, *Proc. OOPSLA '93*, ACM, pp.48-64.

[Kulkarni, 93] D. Kulkarni et. al., *Information Access in Mobile Computing Environments*, Tech. Report 93-11, Dept. of Computer Science & Engg., University of Notre Dame, Notre Dame, Indiana.

[Lajoie, 93] H. Lajoie, Standard C++ Update - The New Language Extensions, *C++ Report*, July-Aug. 93, pp. 47-52.

[OMG, 91] *The Common Request Broker: Architecture and Specification*, OMG Document No. 91.12.1, Object Management Group, Framingham, MA.

[Peterson, 90] L. Peterson, N. Hutchinson, S. O'Malley & H. Rao, The x-kernel: A Platform for Accessing Internet Resources, *IEEE Computer*, 23(5), May 1990, pp. 23-33.

[Skarra, 86] A. Skarra & S. Zdonik, The Management of Changing Types in an Object-Oriented Data Base, *Proc OOPSLA '86*, ACM, pp. 483-495.

[Stroustrup, 91] B. Stroustrup, *The C++ Programming Language*, 2nd Edition, Addison Wesley.

[Stroustrup, 92] B. Stroustrup, D. Lenkov, Runtime Type Identification for C++, *C++ Report*, 4(3), March-April 92, pp. 32-42.

[Ungar, 87] D. Ungar & R. Smith, Self: The Power of Simplicity, *Proc. OOPSLA '87*, pp. 227-242.

[USL, 92] *C++ Language System*, Unix System Labs.

[Vines, 93] D. Vines, Z. Kishimoto, Smalltalk's Runtime Type Support for C++, *C++ Report*, 5(1), Jan. 93, pp. 44-52.

[Yokote, 92] Y. Yokote, The Apertos Reflective Operating System: The Concept and its Implementation, *Proc. OOPSLA '92*, ACM, pp. 414-434.

Implementing Signatures for C++

Gerald Baumgartner Vincent F. Russo
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

gb@cs.purdue.edu russo@cs.purdue.edu

Abstract

In this paper we overview the design and implementation of a language extension to C++ for abstracting types and for decoupling subtyping and inheritance. This extension gives the user more of the flexibility of dynamic typing while retaining the efficiency and security of static typing. We discuss the syntax and semantics of this language extension, show examples of its use, and present and analyze the cost of three different implementation techniques: a preprocessor to a C++ compiler, an implementation in the front end of a C++ compiler, and a low-level back-end based implementation.

1 Introduction

In C++, as in several other object-oriented languages, the *class* construct is used to define a type, to implement that type, and as the basis for inheritance, type abstraction, and subtype polymorphism. We argue that overloading the class construct limits the expressiveness of type abstraction, subtype polymorphism and inheritance. We remedy these problems by introducing a new C++ type definition construct: the *signature*. Signatures provide C++ with a type system that allows for clean separation of interface from implementation and achieves more of the flexibility of dynamic typing without sacrificing the efficiency and security of static typing.

The remainder of the paper is structured as follows. First we present motivation for the addition of a type abstraction facility other than classes to C++. We then present in some detail the syntax and semantics of signatures relative to their implementation. We follow with examples which illustrate how signatures solve the problems presented in the motivation section. The final sections of the paper discuss and compare three different implementation possibilities, and analyze the costs of each. The primary intent of this paper is to detail these implementation techniques. For this reason, the motivation and language specifications are of necessity brief. The reader interested in a more detailed motivation and complete syntax and semantics is referred to [4].

2 Motivation

Using inheritance as a subtyping mechanism suffers from three specific problems:

1. In some cases, it is difficult (if not impossible) to retroactively introduce abstract superclasses for the purpose of type abstraction.
2. Using the same construct (class inheritance) for type abstraction and code sharing limits the power of both and unnecessarily couples implementation to interface.
3. The hierarchy of type abstraction and the class hierarchy of implementation may not easily agree.

We will show how signatures allow us to overcome these problems without a major overhaul of the C++ type system.

2.1 Retroactive Type Abstraction

A practical example of the need to introduce type abstractions of existing class hierarchies is illustrated in [15]. Summarizing their presentation, suppose we have two libraries containing hierarchies of classes for X-Windows display objects. One hierarchy is rooted at `OpenLookObject` and the other at `MotifObject`. Further suppose all the classes in each hierarchy implement a `display()` and a `move()` member function, and that both libraries are supplied in "binary-only" form. Can a display list of objects be constructed that can contain objects from *both* class libraries *simultaneously*? The answer is yes, but not without either explicit type descrimination or substantial software engineering costs due to the introduction of additional classes.

Obviously, the straightforward solution would be to create a common abstract superclass for both hierarchies. However, if no source code is available for the two libraries (only header files and binaries are provided) retroactive code modification is not possible. The only choices remaining are to use a discriminated union for the display list elements, to use multiple inheritance to implement a new set of leaf classes in each hierarchy, or to use a hierarchy of forwarding classes[1]. The former solution is rather inelegant, the latter two clutter up the name space with a superfluous set of new class names.

The problem is that C++ provides only one type abstraction mechanism, the class, and that implementations must explicitly state their adherence to an abstraction by inheriting the abstraction class. The nature of the restrictions in this example prevent us from doing this. What we would like is a type abstraction mechanism which does not rely on classes and, therefore, leaves classes free to be used for implementation specification. Likewise, the adherence of a particular class to a type abstraction would ideally be inferred from the class specification and not need to be explicitly coded in the class. This leaves us free to introduce new type abstractions at a later time without altering any implementations.

2.2 Separation of Type and Class Hierarchies

Another problem with a single class hierarchy defining both abstract data types and their implementations is that as the type hierarchy becomes more complex, it might become necessary to duplicate code, as an example from computer algebra [5, 4] demonstrates.

Consider the abstract type `general_matrix` with subtypes `negative_definite_matrix` and `orthogonal_matrix`. Both subtypes have additional functions, such as `inverse()`, which are not present in general matrices. Assume we have several different implementations of those abstract types, namely `dense_matrix`, which implements matrices as two-dimensional arrays, `sparse_matrix`, which uses lists of triples, and `permutation_matrix`, which is implemented as a special case of sparse matrices that takes advantage of permutation matrices only having one element in each row and column.

If we try to model this example with a class hierarchy, we end up either duplicating code or violating the type hierarchy. While `dense_matrix` can be made a subclass of the abstract virtual classes `general_matrix`, `negative_definite_matrix`, and `orthogonal_matrix` using multiple inheritance, we cannot do the same for `sparse_matrix`. Doing so would make `permutation_matrix`, which is a subclass of `sparse_matrix`, an indirect subclass of `negative_definite_matrix`. Since permutation matrices are positive definite, this would violate the type hierarchy. The alternative of having a separate class `sparse_negative_definite_matrix` is not satisfying either.

Similar arguments have been given in the literature to show that the `collection` class hierarchy of SMALLTALK-80 [14] is not appropriate as a basis for subtyping. While the problem does not arise with dynamic typing, it becomes an issue when trying to make SMALLTALK-80 statically typed while retaining most of its flexibility. The solution is to factor out the implementation aspect of classes into prototypical objects [18] or to factor out the type aspect into interfaces [7, 9].

[1] In C++, the task of creating these leaf and forwarding classes can be simplified using templates.

2.3 Implementation of Conflicting Type and Class Hierarchies

Often the abstract type hierarchy and the implementation class hierarchy cannot be made to agree. An example similar to one in [22] illustrates this point. Consider two abstract types **queue** and **dequeue** (doubly ended queue). The abstract type **dequeue** provides the same operations as **queue** and two additional operations for enqueuing at the head and for dequeuing from the tail of the queue. Therefore, **dequeue** is a *subtype* of **queue**.

However, the easiest way to implement **queue** and **dequeue** is to structure the inheritance hierarchy opposite to the type hierarchy. A doubly ended queue is implemented naturally as a doubly linked list. A trivial implementation of queue would be to copy the doubly ended queue implementation through inheritance and remove, or ignore, the additional operations.

In [10] it is argued that in order for a type system to be sound it should not be possible to use inheritance for subtyping purposes and also allow the removal of operations. Most object-oriented languages choose instead to restrict the use of inheritance for code sharing to situations where there is also a subtype relationship, and to disallow inheriting only a portion of the superclass.

3 Syntax and Semantics of the Signature Language Extension

We term the key language construct we add to C++ to support type abstraction a *signature*. It is related to types in RUSSEL [11], ML's signatures [19, 20], HASKELL's type classes [16], definition modules in MODULA-2 [26], interface modules in MODULA-3 [8], abstract types in EMERALD [6], type modules in TRELLIS/OWL [21], categories in AXIOM [17] and its predecessor SCRATCHPAD II [24, 25], and types in POOL-I [3].

The type system of C++ with signatures comes closest to those of AXIOM and POOL-I. RUSSEL, ML, HASKELL, and MODULA-2 don't have class types, MODULA-3 only has interfaces for modules but not for classes. EMERALD has first-class types instead of classes, and TRELLIS/OWL has a type hierarchy in which type information but no implementation is inherited. Domains in AXIOM differ from classes by having method dispatch on all argument types and on the return type. Compared to C++, POOL-I doesn't have private and protected member functions and overloading. While both categories and domains in AXIOM and types in POOL-I are first class, signatures and classes in our C++ extension are not, which makes the type system slightly less expressive but allows for a more efficient implementation and for complete type checking at compile time.

In this section we describe only those parts of our language extension that are relevant to contrasting the different implementation techniques discussed later in the paper. Specifically, this section details the syntax and semantics of signatures, signature pointers, and signature references. We also explain the semantics and utility of default implementations, views, and constants in signatures[2].

3.1 Signature Declarations

A signature type is declared in a way similar to a class except the keyword **signature** is used instead of **class**, or **struct**, to introduce a signature declaration.

A signature declaration, like a class declaration, defines a new C++ type. The key difference is that a signature declaration contains only *interface descriptions*. For example, the signature declaration

```
signature T {
    int * f ();
    int g (int *);
    T & h (int *);
};
```

[2]The additional features of signature inheritance, the **sigof** construct (as in [15]), and opaque types are left out since they only affect the type checking phase of the compiler. For information on those constructs, as well as for more details on the semantics of signatures, see [4].

defines an abstract type `T` with operations (member functions) `f`, `g`, and `h`.

The specific difference from a class declaration is that only type declarations (`typedefs`), constant declarations, member function declarations, operator declarations, and conversion operator declarations are allowed within a signature declaration. Specifically:

- A signature cannot have constructors, destructors, friend or field declarations.
- The visibility specifiers `private`, `protected`, and `public` are not allowed either in the signature body or in the base type list. They are unnecessary since signatures define interfaces and, therefore, have all "public members" implicitly.
- Signature base types have to be signatures themselves (a signature cannot inherit from a class). Similarly, a signature cannot be the base type of a class.
- The type specifiers `const` and `volatile` are not allowed for signature member functions, since they are storage location specifiers and are meaningless for members of an interface specification.
- The storage class specifiers (`auto`, `register`, `static`, `extern`), the function specifiers `inline` and `virtual`, and the pure specifier `=0` are not allowed. The latter two are needed in class declarations only to specify abstract classes and are, therefore, superfluous in signature declarations.

In the absence of a more complex type hierarchy, the type `T` in the above example could have been defined as an abstract class, i.e., a class containing only pure virtual member function declarations [12]. The behavior of both implementations would be similar except that classes implementing the abstract class's interface need to explicitly code that fact by inheriting from the abstract class. When using signatures to specify abstract types, this relationship can be inferred by them compiler.

As a type hierarchy becomes more complex it becomes more and more difficult to model it precisely with a class hierarchy as shown in the computer algebra example. With signatures, a type hierarchy structured independently from the class hierarchy can be built. This enables more complex type hierarchies and facilitates the decoupling of subtyping and inheritance. Also, signatures can be used to define type abstractions of existing class hierarchies. With abstract classes, it would be necessary to retrofit abstract classes on top of the existing class hierarchy. This cannot be done without recompiling all existing source files. Signatures, therefore, improve C++'s capabilities for reusing existing code.

3.2 Signature Pointers and References

Since a signature type declaration only describes an abstract type, it does not give enough information to create an implementation for that type. For this reason it is not valid to declare objects of a signature type, as in

```
signature S { /* ... */ };
S o;   // illegal! 'S' is an interface type
```

Instead, in order to associate a signature type with an implementation, we declare a *signature pointer* or a *signature reference* and assign to it the address of an existing class object. Signature pointers and signature references, therefore, can be seen as *interfaces* between abstract (signature) types and concrete (class) types.

Consider the following declarations,

```
signature S { /* ... */ };
class C { /* ... */ };
C o;
S * p = &o;  // legal if 'C' conforms to 'S'
```

For the initialization of the signature pointer `p`, or for an assignment to `p`, to be type correct, the class type `C` has to *conform* to the signature type `S`. I.e., the implementation of `C` has to satisfy the interface `S`, or the signature of `C` has to be a sub*type* of `S`.

A signature pointer or reference can also be assigned to another signature pointer or reference. In this case, the right hand side signature must conform to the left hand side signature, or in other words, the right hand side signature must be a subtype of the left hand side signature.

3.3 The Conformance Check

The *conformance check* is the type check performed when initializing or assigning to a signature pointer or a signature reference. Except for the very rare case described below, the design and implementation of signatures implies no *run-time* cost for the conformance check, the checking is done solely at *compile* time.

To test whether a class `C` conforms to a signature `S`, the structures of `C` and `S` must be recursively compared. The specific conformance rules are:

1. For every member function, operator, and conversion operator declared in `S`, there must be a public declaration of the same function or operator in `C`. Furthermore, this declaration must have the same name and conforming return and argument types. Also, every signature contains an implicit destructor declaration. This destructor is matched with the class's destructor if defined or with the default destructor otherwise. Specifically, a signature member function `S::f` conforms to a class member function `C::f` iff
   - The type of every argument of `S::f` conforms to the type of the corresponding argument of `C::f` and
   - The return type of `C::f` conforms to the return type of `S::f`.
2. For every type definition in `S`, there is a public type definition of the same name and conforming structure in `C`.
3. For every constant declaration in `S`, there is a constant declaration of the same name and conforming type in `C`.

The conformance check for testing the conformance of one signature to another is exactly the same.

Field declarations as well as private or protected member functions and constructors in `C` are ignored during conformance checking. Also, `C` can have more public member functions or types than those specified in `S`.

For example, suppose we are testing the conformance of class `C` to signature `S`. Given signatures `T` and `U` and classes `D` and `E`, let signature `U` conform to signature `T`, let class `D` conform to signature `T`, and let class `E` be a subclass of class `D`. The signature member function

```
T * S::f (D *, E *);
```

can be matched with any of the following class member functions:

```
T * C::f (D *, E *);      // since the types are the same
T * C::f (D *, D *);      // since 'D' is a supertype of 'E'
T * C::f (T *, E *);      // since 'D' conforms to 'T'
T * C::f (T *, T *);      // since both 'D' and 'E' conform to 'T'
D * C::f (D *, E *);      // since 'D' conforms to 'T'
E * C::f (D *, E *);      // ...
U * C::f (D *, E *);
T * C::f (D *, E *, int = 0);
```

Note that conformance (and therefore subtyping) is defined using contravariance of the argument types of member functions unlike the subtype relationship defined by class inheritance. This is

necessary to make subtyping sound and to avoid run-time type errors as they can occur in C++'s inheritance-based subtyping.

If several member functions of `C` conform to one member function of `S`, we find the one that conforms best using a variant of C++'s algorithm for finding the function declaration that best matches the call of an overloaded function [12].

If a member function of `C` conforms to several member functions of `S`, an error is reported by the compiler. This restriction could be relaxed by considering different matches of `C`'s member functions with `S`'s member functions and by picking the best match according to some metric on signature types. However, we feel that any such rule would be sufficiently complex to confuse users.

3.4 Default Implementations

Since signature declarations declare interface types, they usually only contain member function and operator *declarations*. However, a signature declaration can also contain member function *definitions* (i.e., declarations together with implementations). Such definitions are called *default implementations*. Consider, for example, the signature

```
signature S {
    int f  (int);
    int f0 (void) { return f (0); };
};
```

For a class `C` to conform to `S`, it is not necessary for `C` to contain the member function '`int f0 (void)`.' However, if `C::f0` is defined and of the right type, it will be used. If `C::f0` is not defined the default implementation `S::f0` is used instead.

Default implementations are useful for rapid prototyping during interface design since they allow quick implementations of functions and classes which can later be replaced by more efficient or sophisticated implementations. For example, a design could define an **`integer`** signature with addition and multiplication member functions, and implement it with a class which only supports addition. Multiplication could be implemented in the signature by a default member function which does repeated additions. In the later stages of the design, a class with a member function that does multiplication directly can be added without changing any other code.

One consequence of allowing default implementations is that they introduce a case that cannot be type checked fully at compile time. The problem arises when assigning a signature pointer of signature type `T` to a signature pointer of signature type `S`, where `T` contains a default implementation for a member function `f` but `S` only contains a *declaration* of `f`. Since it is not known at compile time whether the default implementation of `T::f` is actually used, a run-time test for it must be generated. Consider

```
signature S {
    int f (void);
};

signature T {
    int f (void) { return 0; };
};

int foo (T * p)
{
    S * q = p;

    /* ... */
}
```

In the function `foo` above it cannot be known whether `p` will use `T`'s default implementation or not. If the default implementation is used, there will be a run-time type error in the assignment to `q`.

Note that this is the only case where a run-time type check is necessary, in all other cases conformance can be fully checked at compile time. To warn of the possibility of a run-time type error, our compiler prints a warning message when generating the run-time test.

3.5 Views

The earlier presentation of the signature conformance check required that member functions declared by a signature be matched by class member functions of the same name and the same type. This may not be completely realistic as it might be the case that the implementor of the class simply chose a different name for the same function.

For example, suppose that in the X-Windows object manager example the function to display a window on the screen is called `display()` in `OpenLookObject` but `show()` in `MotifObject`. To build a display list of objects from both hierarchies, it would be necessary to rename the member function in one of those hierarchies.

Such renaming could be partially achieved through default implementations, but this is not sufficiently powerful to, for example, swap the names of two member functions. Instead, to rename class member functions, or to *view* a class to be an implementation of a signature type in a different way, we provide the following syntax:

```
S * p = (S *; foo = bar) new C;
```

This expression associates the class member function `bar` with the signature member function `foo`. Renaming expressions can be separated by commas in order to rename multiple member functions.

Conceptually, the renaming operations are performed in parallel to allow swapping of member function names. This allows, for example, a rational number class to be viewed as an implementation of the abstract type `Group` in two different ways, as a multiplicative group and as an additive group.

If the renamed member function is overloaded, all overloaded definitions are renamed in the same way. There is no syntax for selectively renaming functions depending on their return and argument types. While this would be possible, we feel it would make the syntax of views too complicated.

A similar renaming mechanism can be found in the computer algebra system VIEWS implemented in SMALLTALK [1, 2] or in the algebraic specification language OBJ3 [13].

3.6 Constants

As mentioned in the definition of the conformance check, a signature can contain constant declarations. Unlike constant declarations elsewhere, constants in signatures need not be initialized. Instead, they are treated as nullary functions. For example, a class conforming to

```
signature S {
    const int n;
};
```

has to have a public declaration of constant `n`. The value of the class's constant can then be accessed through a signature pointer as in the following example.

```
class C {
 public:
    const int n = 17;
};

S * p = new C;
int i = p->n;
```

The variable `i` above gets the value `17`. The behavior is the same as if the constant `n` had been replaced by a nullary function returning the constant value, except that it can be implemented more efficiently.

It is possible to implement *initialized* constants in signatures, and treat them like constant nullary functions with a default implementation, i.e., the value of the class's constant overrides the value of the signature's constant. However, since we also want to use constants for defining data structures, we require that the value of a constant in both the class and the signature is the same. Otherwise, it would be impossible to write code such as

```
signature S {
    const int n = 17;
    typedef int[n] array;
    int f (array);
};
```

since the value of `n` would not be known at compile time.

4 Example Uses of Signatures

4.1 Signatures for Retroactive Type Abstraction

The solution to the `XWindowsObject` example using signatures is actually quite simple. All that is needed is to introduce a signature to define the abstract type `XWindowsObject`,

```
signature XWindowsObject {
    void display (void);
    void move    (void);
};
```

and to implement the display list as a list of pointers `XWindowsObject`'s,

```
XWindowsObject * displayList[NELEMENTS];
```

Given a pair of implementation hierarchies such as:

```
class OpenLookObject {
 public:
    virtual void display (void);
    virtual void move    (void);
    // ...
};
```

and

```
class MotifObject {
 public:
    virtual void display (void);
    virtual void move (void);
    // ...
};
```

It is simple to use the display list. For example,

```
int main (void);
{
    displayList[0] = new OpenLookCircle;
    displayList[1] = new MotifSquare;
    // ...

    displayList[0]->display ();          // executes OpenLookCircle::display
    displayList[1]->display ();          // executes MotifSquare::show
```

```
    return 0;
}
```

where `OpenLookCircle` is a subclass of `OpenLookObject` and `MotifSquare` is a subclass of `MotifObject`.

We can even make this example more compelling. Consider the possibility that the Motif class hierarchy used the name `show` rather than `display` for its rendering operation. We would simply need to add a view cast when assigning an object from the Motif hierarchy to our display list:

```
displayList[1] = (XWindowsObject *; display = show) new MotifSquare;
```

4.2 Signatures to Separate Type and Class Hierarchies

The solution to model the type and implementation hierarchies in the computer algebra example is to use signatures instead of abstract virtual classes for the type hierarchy:

```
signature general_matrix           { /* ... */ };
signature negative_definite_matrix { /* ... */ };
signature orthogonal_matrix        { /* ... */ };
```

Since `negative_definite_matrix` and `orthogonal_matrix` conform to `general_matrix` they are also subtypes of `general_matrix`. By using inheritance of signatures, as defined in [4], we can simplify the definition of the latter two signatures.

For modelling the implementation hierarchy we use classes and class inheritance:

```
class dense_matrix  { /* ... */ };
class sparse_matrix { /* ... */ };
class permutation_matrix : sparse_matrix { /* ... */ };
```

Signature conformance ensures that we can use these classes as implementations of the above signature types. Note that we use private inheritance for defining `permutation_matrix`. This allows us to hide any member functions defined in `negative_definite_matrix` but not in the other two signatures.

4.3 Signatures to Implement Conflicting Type and Class Hierarchies

The solution to the queue/dequeue problem presented earlier is also quite easy using signatures. Simply define an implementation class, and two signatures to define the abstract types `queue` and `dequeue`.

```
template <class T> class DoublyLinkedList {
public:
    void enqueueHead( T );
    T dequeueHead();
    void enqueueTail( T );
    T dequeueTail();
    // ...
};

template <class T> signature dequeue {
    void enqueueHead( T );
    T dequeueHead();
    void enqueueTail( T );
    T dequeueTail();
};
```

```
template <class T> signature queue {
    void enqueueTail( T );
    T dequeueHead();
};

queue<int> * q1 = new DoublyLinkedList<int>;
dequeue<char *> * q2 = new DoublyLinkedList<char *>;
```

It should be noted that this same effect can be achieved in C++ without signatures by using multiple inheritance. E.g., by implementing `queue` and `dequeue` as abstract classes and having `DoublyLinkedList` inherit from both. To see where this type of solution breaks down, consider adding another type, `stack`, with `push` and `pop` members. With signatures it is simple to define a `stack` signature and whenever assigning a `DoublyLinkedList` use a view cast to remap `push` to `enqueueHead` and `pop` to `dequeueHead`. With the multiple inheritance based solution, it would be necessary either to introduce a new multiply inherited abstract class that *implements* `push` and `pop` by delegating to `enqueueHead` and `dequeueHead`, or to alter `DoublyLinkedList` to implement `push` and `pop` directly. The former unnecessarily constrains the implementation of other classes that might implement the stack abstraction, while the later needlessly clutters the implementation of `DoublyLinkedList`.

5 Implementation Techniques

In this section, we present three options for implementing signatures. The first method could be used in a compiler preprocessor (e.g., a `cfrontfront`) that translates C++ with signatures into C++ without signatures. The second is a compiler based implementation that produces a C-level code version of signatures and needs direct access to the type checking phases of a C++ compiler, but is independent of the compiler back-end and machine architecture. This method has been implemented in the GNU C++ compiler[23] as a modification of GCC's C++ front end, `cc1plus`. The same techniques are equally applicable to AT&T's `cfront`, or other C++ compilers. Finally, we outline an implementation technique that requires knowledge of the compiler back-end and code generation phases to generate assembly-level code to further optimize signature member function calls.

5.1 Preprocessor-Based Implementation

The main idea of implementing signatures is to generate interface objects that encapsulate the class objects. These interface objects forward the signature member functions to the appropriate class member functions. Signature pointers can then be implemented as regular C++ pointers that point to those interface objects.

Consider the declarations

```
signature S {
    int f (void);
    int g (int, int);
};

S * p = new C;
```

and assume `C` conforms to `S`. The signature declaration itself is simply a type declaration, we do not need to generate any code for it. The code for the interface object is generated when compiling the assignment to the signature pointer `p`.

In the particular case above, the interface object must redirect the signature member functions `S::f` and `S::g` to the corresponding class member functions `C::f` and `C::g`.

To create such interface objects for any class `C` that conforms to a signature `S`, we first generate an abstract virtual class `S_Interface`. For each class `C`, we then need a subclass of `S_Interface` that redirects the signature member functions to the class member functions of the given class.

For the signature `S` given above, we generate the following abstract virtual class:

```
class S_Interface {
 public:
    virtual ~S_Interface ()  = 0;
    virtual int f (void)     = 0;
    virtual int g (int, int) = 0;
};
```

For creating the classes of interfaces objects, we generate a template class `S_C_Interface` as public subclass of `S_Interface`.

```
template <class C> class S_C_Interface : public S_Interface {
    C * optr;
 public:
    S_C_Interface (C * q)  { optr = q; };
    ~S_C_Interface (void)  { delete optr; };
    int f (void)           { return optr->f (); };
    int g (int x, int y)   { return optr->g (x, y); };
};
```

This template class is then instantiated with some class `C` to build the class of objects interfacing `S` and `C`.

Signature pointers can now be implemented as pointers to objects of type `S_C_Interface<C>` for a given class `C`. That is, the declaration

```
S * p = new C;
```

is translated to

```
S_Interface * p = new S_C_Interface<C> (new C);
```

Since a signature pointer is a standard C++ pointer in this scheme, we don't need to do anything special to compile a signature member function call. The call `p->f ()` will execute `S_C_Interface<C>::f`, which in turn calls `C::f`.

To compile signature constants without initialization, the constant must be translated into a variable in the interface class. Assume our signature `S` contains the constant declaration '`const int c;`.' We translate this declaration into the private member declaration '`int c;`' in class `S_Interface`, and initialize `c` in the constructor of the template class `S_C_Interface`:

```
S_C_Interface (C * q, int i)  { optr = q;  c = i; }
```

For initializing a signature pointer or for assigning to one, the value of the class constant has to be provided as the second argument of the constructor:

```
S_Interface * p = new S_C_Interface<C> (new C, C::c);
```

To implement default implementations, we have to add a flag to the interface object that tells us whether a given member function is provided by the class or not. Assume that the signature member function `f` comes with a default implementation. We add the flag '`unsigned int f_flag:1;`' to class `S_Interface` and generate the following code for the member function `f` in class `S_C_Interface`:

```
int f (void)
{
    if (f_flag)
        return optr->f ();

    // code for default implementation
}
```

Similarly as with signature constants, the flag has to be initialized in `S_C_Interface`'s constructor.

The above solution has the advantage that it is straightforward to implement in a preprocessor for a C++ compiler. The disadvantage is that it requires interface objects to be allocated on the heap. To avoid heap allocation, we can use the interface object itself as a signature pointer. In this case, the declaration of `p` is translated to

```
S_C_Interface<C> p = new C;
```

This solution requires some more intelligence in the preprocessor to make `p` behave as if it were a pointer of type `S_Interface *`. For example, the signature member function call `p->f ()` now needs to be translated into `p.f ()`. Signature references are implemented exactly the same way as signature pointers.

For signatures that don't have default implementations or constants, the storage needed for an interface object is two words, the pointer to the class object, `optr`, and the pointer to `S_C_Interface<C>`'s virtual function table. Each default implementation requires one additional bit, and constants can be arbitrarily large. Therefore, performing the above optimization for reducing heap allocation should be conditional on the size of the interface object. With signature pointers being the interface objects themselves, assigning one signature pointer to another requires copying the entire structure. If the signature pointer takes only two words of storage, copying is not a problem. With a constant array of several kilobytes in a signature, copying is certainly a bad choice.

5.2 Compiler Front-End Implementation

In the above implementation, there are two sources of inefficiency, which we try to overcome in our GNU C++ implementation. One of the problems is that for calling a signature member function, we have two member functions calls, the call to the interface object's member function and the call to the class member function. The other problem is that interface objects can become rather large.

In order to optimize signature member function calls we can inline the member function calls of class `S_C_Interface<C>` by storing some of the information contained in those member functions in a special table called the *signature table*. A signature table, which is similar in structure to a virtual function table, depends only on a signature and conforming class pair, and can, therefore, be made static.

The key to optimizing the space requirements of interface objects is to observe that signature constants, as well as the default implementation flags, can be stored in static memory as well. The values of both signature constants and default implementation flags can be determined in the class conformance check, they don't depend on the actual class object. The obvious place to store them is, again, in the signature table.

To keep the structure of signature tables in our implementation simpler, we impose a slight restriction on the expressiveness of the language by requiring *strict conformance* between a signature and a class. Strict conformance means that a signature member function and the corresponding class member function need to have the same number of arguments, exactly the same argument types, and exactly the same return type. Without imposing this restriction, we might need to convert argument types or the return type in a signature member function call, but we don't have place in a signature table to store the conversion code. In a following section, we will show how to lift this restriction without causing run-time overhead.

Simplified Version

Let us ignore default implementations, signature constants, classes with virtual member functions, and multiple inheritance of classes for now. For the above signature declaration with member functions `f` and `g`, the compiler generates an internal representation of the following structure of member function pointers:

```
struct S_Table {
    const void * _S_destr;
    const int    (S::*_f) (void);
```

```
    const int    (S::*_g) (int, int);
};
```

where the field `_S_destr` represents the destructor that is implicitly declared in every signature. The type `S_Table` will be the type of signature tables for signature `S`.

In the previous solution, an interface object contained a pointer to the class object and a pointer to a virtual function table. In this scheme, we have a pointer to the signature table instead of the virtual function table pointer, and we store the interface object in the signature pointer. This leads us to the following type declaration for signature pointers:

```
struct S_Pointer {
    void *          optr;
    const S_Table * sptr;
};
```

Actually, the type of `optr` can be a pointer to *any* object. When using `optr` the compiler must generate appropriate casts.

We now generate code for the declaration '`S * p = new C;`' as follows:

```
static const S_Table S_C_Table = { &C::~C, &C::f, &C::g };
S_Pointer p = { new C, &S_C_Table };
```

To initialize the signature table `S_C_Table`, we need to cast `C::f` and `C::g` to member functions of `S`. If `C` doesn't have a destructor, we use the default destructor. Since C++ doesn't allow us to cast to a member function type or to take the address of a destructor, this has to be done in the compiler front end.

While we can use a default constructor for initializing a signature pointer as shown above, we need to translate an assignment to a signature pointer into a compound expression. For the assignment expression '`p = new C`' or for passing an object to a signature pointer parameter in a function call, the compiler generates the compound expression

```
( p.optr = new C,
  p.sptr = &S_C_Table,
  p
)
```

as well as the declaration and initialization of the signature table:

```
static const S_Table S_C_Table = { &C::~C, &C::f, &C::g };
```

If the assignment was in an inner scope, the signature table declaration needs to be moved out of this scope into the file scope.

Since signature tables are static and constant, only one signature table declaration needs to be constructed in each file per signature-class pair.

To compile a function call such as

```
int i = p->g (7, 11);
```

we need to dereference `p`'s `sptr` and call the function whose address is stored in the field `_g`, which is `C::g` in our example. We need to pass the value of `p`'s `optr` field as first argument, so that `C::g` gets a pointer to the right object passed for its implicit first parameter called `this`.

```
int i = p.sptr->_g (p.optr, 7, 11);
```

If the compiler knows the current value of `p->sptr`, this can be optimized to a direct call to `C::g`.

Fine Details

As noted earlier, we need to store signature constants and flags to test for default implementations in the signature table. Similarly, we need additional information to handle multiple inheritance of classes and classes with virtual member functions.

If a signature member function is implemented by a virtual class member function, we don't know the address of the function to call until run time. It is, therefore, not possible to store the class member function's address in the signature table. What we do instead is store the offset into the class's virtual function table together with a flag that tells us to perform a virtual member function call. Together with calls to default implementations, we have now three different kinds of signature member function calls. We store the two flags needed to test for those cases in a short integer, which will be tested for a zero, positive, or negative value.

In case of multiple inheritance, an object in GCC might use several virtual function tables, one for each parent class. We need to find the right virtual function table to use for each member function call. The solution is to add another field to the signature table entry that contains the offset into the object where we can find a pointer to the proper virtual function table.

Also in GCC, member functions are implemented as regular functions that take a pointer to the object, called `this`, as first argument. To pass the object correctly in the presence of multiple inheritance of member functions, an offset has to be added to `this` that depends on the place of the member function in the class hierarchy. As in virtual function tables, we need to store this offset with each entry in a signature table.

To summarize, a signature table entry has the following structure:

```
struct sigtable_entry_type {
    short  code;
    short  offset;
    union {
        void * pfn;
        struct {
            short vpoffset;
            short vtoffset;
        };
    };
};
```

The `code` field contains the flags mentioned above, the `offset` field contains the value to be added to `this`, `pfn` contains a function pointer in case of a non-virtual member function or a default implementation, and, in case of a virtual member function, `vpoffset` and `vtoffset` contain the offset of the virtual function table pointer in the object and the offset into the virtual function table, respectively. The fields `vpoffset` and `vtoffset` occupy the same memory location as `pfn`. For type checking purposes, the compiler needs to cast `pfn` to the appropriate function pointer types.

The signature table is a structure that contains a field of type `sigtable_entry_type` for every member function declared in the signature and for the implicitly declared destructor. For signature `S` declared earlier, the signature table looks like:

```
struct S_Table {
    const sigtable_entry_type _S_destr;
    const sigtable_entry_type _f;
    const sigtable_entry_type _g;
};
```

In addition, for each uninitialized constant in the signature, we insert a field declaration into the signature table type. All the information for initializing the fields of a signature table entry and for initializing constants can be obtained at compile time from the class of the object on the RHS of a signature pointer assignment or initialization.

When assigning a signature pointer to another signature pointer of the same type, we simply copy the two fields. If the types are not the same, we may need to initialize the signature table

for the LHS signature pointer at run time by copying the corresponding fields from the signature table to which the RHS signature pointer points. If the signature table entries to be copied form a contiguous block of data in the RHS signature table and the order of the table entries is the same for both signature tables, the compiler can avoid copying by letting the LHS `sptr` point into the RHS signature table.

If copying is unavoidable, the compiler should print a warning message. Independent of whether copying table entries is necessary, if the RHS signature contains a default implementation where the LHS signature only has a member function *declaration*, the compiler needs to generate a run-time test and should print a corresponding warning message.

To call a signature member function, we need to generate a conditional expression that tests the `code` field of the signature table entry and, depending on its value, call a non-virtual function, a virtual function, or a default implementation. We also have to make sure that the right offset gets added to the `this` pointer. The signature member function call

```
int i = p->g (7, 11);
```

from our example above is now translated into

```
int i = (this = p.optr,
         s = p.sptr->_g,
         s.code == 0
             ? s.pfn (this + s.offset, 7, 11)        // non-virtual call
             : (v = (*p.optr[s.vpoffset])[s.vtoffset],
                v.pfn (this + s.offset, 7, 11)       // virtual call
               )
        );
```

where `this`, `s`, and `v` are compiler generated temporary variables to hold the pointer to the object, the signature table entry, and the virtual function table entry, respectively. These temporary variables can be kept in registers.

In case the signature member function `g` has a default implementation, the signature member function call becomes

```
int i = (this = p.optr,
         s = p.sptr->_g,
         s.code == 0
             ? s.pfn (this + s.offset, 7, 11)        // non-virtual call
             : (v = (*p.optr[s.vpoffset])[s.vtoffset],
                s.code > 0
                    ? v.pfn (this + s.offset, 7, 11) // virtual call
                    : s.pfn (p, 7, 11)               // default impl call
               )
        );
```

Since in practice a non-virtual function call is the most common case, it should be reached with only one test.

5.3 Implementation with Back-End Support

A place that leaves room for optimization in the previous solution is the conditional expression for calling a signature member function. A possible way to optimize signature member function calls is to store (pointers to) pieces of code, or *thunks*, in the signature table instead of flags and offsets. The thunks contain the appropriate code to set the `this` pointer correctly and branch to the class member function or perform a virtual function call. Such an implementation was proposed in [15].

Each thunk only contains the code necessary to call one specific class member function. We do not need to test any flags but just branch to the thunk, which does the right thing for the member function we want to call. Signature table entries are now reduced to a single function pointer again.

For example, given the signature `S` with member functions `f` and `g` as above, the signature table is of type

```
struct S_Table {
    void * _f;
    void * _g;
};
```

Given a class `C` conforming to `S`, assume that `C::f` is a non-virtual member function that doesn't need any offset to be added to `this` and that `C::g` is a virtual member function that requires adding a non-zero offset of `this`. The thunk needed is the following short piece of code:

```
S_C_f_Thunk:
{
    this = this.optr;
    goto C::f;
}
```

Before branching to the thunk, the compiler will have set up the activation record correctly for calling `C::f`. In particular, all the arguments were either pushed onto the stack or are in registers. The value passed for the first argument, `this`, is the signature pointer. Before branching to `C::f`, we need to extract the `optr` field so that `this` points to the object. With the right layout of the activation record in registers or on the stack, no work needs to be done for adjusting `this`. In that case we can get rid of the thunk altogether and store a direct pointer to `C::f` in the signature table entry.

For the virtual member function `C::g` we need the thunk

```
S_C_g_Thunk:
{
    temp = this;
    this = this.optr + OFFSET;
    goto (*(temp.optr)[VPOFFSET])[VTOFFSET].pfn;
}
```

The values `OFFSET`, `VPOFFSET`, and `VTOFFSET` are constants that can be determined at compile time and are hard-coded into the thunk.

When a default implementation is used, the entry in the signature table can point to the code of the default implementation directly; we don't need a thunk in this case.

When compiling an assignment of an object of class `C` to a signature pointer, the compiler generates the above thunks and generates a declaration of the signature table,

```
static const S_Table S_C_Table = { &S_C_f_Thunk, &S_C_g_Thunk };
```

and initializes it to point to the thunks. If a default implementation is used, the corresponding signature table entry contains a pointer to the code of the default implementation.

Instead of a conditional expression, the signature member function call

```
int i = p->g (7, 11);
```

now reduces to

```
int i = p.sptr->_g (p, 7, 11);
```

A big advantage is that we could include code for converting argument types in a thunk. This code would simply go into the thunk before the `goto`. Since a signature table is unique for each signature-class pair, the compiler can generate the conversion code for each thunk when generating the signature table. For converting the return type we could either have a second thunk which does nothing else but perform the necessary conversion, or we could call, instead of branching to, the class member function from the thunk using a light-weight function call sequence. The thunk for the non-virtual member function call could look then as follows:

```
{
    this = this->optr + offset;
    // convert argument types
    temp = ret_addr;
    ret_addr = L;
    goto C::f;
 L: // convert return type
    ret_addr = temp;
    return;
}
```

By making signature tables contain thunks we can implement the conformance check completely, i.e., we are not limited to strict conformance. There are no run-time penalties compared to the front-end implementation if a signature member function doesn't require conversions. On the contrary, by not having to test a `code` field as in our implementation, a few instructions will be saved. The only disadvantage of this solution is that it requires generation of low-level code, which complicates or even prohibits its use in a compiler that generates C code, such as AT&T's `cfront` compiler.

As in the front-end implementation, assigning a signature pointer to another signature pointer might require copying entries of the RHS signature table to the LHS signature table. In most cases we can copy the pointer to the thunk. If a member function of the LHS signature does not have the exact same argument and return types as the member function of the RHS signature, however, the compiler needs to generate a new thunk that performs the conversions needed and then branches to the thunk from the RHS signature table, which might do some further conversions.

In the thunk implementation described in [15], copying of signature table entries is avoided by having the `optr` field of the LHS signature pointer point to the RHS signature pointer instead of pointing to the object. This makes assignment more efficient but requires multiple indirections in a signature member function call. Furthermore, to allow assigning a local signature pointer to a non-local signature pointer the solution in [15] has to be corrected and signature pointers have to be allocated on the heap.

There is one more detail in assigning a signature pointer to another signature pointer. If the RHS signature table contains a default implementation that is not allowed to be copied to the LHS signature table, an error has to be reported. To allow this run-time test, we have to reintroduce a flag that tells us whether a default implementation is used or not. This flag can be stored in the low-order bit of the function/thunk pointer in the signature table. We just have to make sure that class member functions and thunks are aligned on half-word or word boundaries, which is required on most RISC-based architectures anyway. When calling a signature member function that might use a default implemention, we have to mask out this bit. If the architecture allows, we could omit the mask instruction by starting default implementations at odd addresses. The only time this flag needs to be tested is in the code for an assignment when performing the run-time error check.

As a further optimization of signature member function calls, the signature table can contain the code for thunks directly instead of a pointer to a thunk. If a thunk contains conversion code and doesn't fit into the allocated space, the signature table would contain a branch instruction to jump to the thunk. This makes signature member function calls more efficient for the most common cases. What would become inefficent, however, is copying signature table entries when assigning one signature pointer to another. To avoid that, this optimization could be controlled by the user, or restricted to the case where the compiler determines that no copying of signature table entries is necessary in the entire source file.

6 Cost Comparison

Ignoring default implementations and constants, the memory required for interface objects in the preprocessor implementation is two words, one for the `optr` field and one for the virtual function table pointer. This is the same size as the size of signature pointers in the other two implementations, where we have an `sptr` field instead of the virtual function table pointer. In interface objects,

we need additional space for constants and default implementation flags. In the compiler-based implementations, the extra space is needed in the signature tables, which are in static memory.

The space needed for the signature table in the compiler front-end implementation is the same as the space needed for the virtual function table in the preprocessor implementation, two words for each signature member function and an additional one for the implicitly declared destructor. Additional space is needed in signature tables for constants. In the thunk implementation, the signature table takes only half the space since we only need one pointer per table entry. But in addition we need static storage for the thunks.

Assigning a class object to a signature pointer requires two pointer assignments in the compiler-based solutions. In the preprocessor implementation, we need to call the constructor of the template class `S_C_Interface`, allocate the interface object on the heap, and then assign two pointers.

Assigning a signature pointer of a different type than the LHS signature pointer can become expensive in the compiler based implementations if signature table entry fields have to be copied. In the preprocessor implementation, we have the same cost as for assigning a class object.

In the preprocessor implementation, a signature member function call takes as much time as two class member function calls, one of which is virtual.

When calling a non-virtual member function in the front-end based implementation, we need one test in addition to the time needed for a virtual member function call. If the `optr` field of the signature pointer is in the wrong register, we also need a register-to-register move. Calling a virtual member function through a signature pointer requires two table lookups, one to get the signature table entry and another to get the virtual function table entry. In addition, there are one or two tests, depending on whether a default implementation might be called or not. The cost of calling a default implementation is two tests added to the cost of a virtual member function call.

In the thunks implementation we don't need to perform any tests. Assuming the register layout is such that the `optr` field of the signature pointer is in the right register to be passed on to the class member function, we can make a signature member function call *exactly* as efficient as a standard virtual member function call in the case of calling a non-virtual member function or a default implementation. When calling a virtual member function through a signature pointer, we have to perform an additional table lookup.

To optimize calling a virtual class member function through a signature pointer, we could copy the information from the virtual function table into the signature table when the member function is called for the first time. This would add the cost of copying in the first call but would make subsequent calls more efficient. We could use this optimization for both compiler based implementations.

7 Conclusion

In this paper, we discussed the limitations of inheritance for achieving subtype polymorphism and for code reuse. We proposed language constructs for specifying and working with abstract types that allow us to decouple subtyping from inheritance, gave the syntax and semantics of such an extension, and proposed three possible implementation strategies for this language extension.

While we presented the ideas of such a language extension as an extension of C++, they would equally well apply to any statically typed object-oriented programming language. Having decoupled subtyping from inheritance, it would be possible to change the semantics of inheritance and make it more versatile for code reuse by allowing to inherit only parts of a superclass while giving up its use for subtyping. For pragmatical reasons, however, such a change is undesirable as it might affect the behavior of existing programs.

8 Availability

Parts of the language extension have already been implemented in GCC; the implementation is available by anonymous `ftp` from `ftp.cs.purdue.edu`, directory `pub/gb`. We expect our extension to become part of the GCC distribution starting with GCC version 2.6.

References

[1] S. Kamal Abdali, Guy W. Cherry, and Neil Soiffer. "An object-oriented approach to algebra system design." In Bruce W. Char (ed.): *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation (SYMSAC '86)*, Waterloo, Ontario, Canada, 21–23 July 1986, pp. 24–30. Association for Computing Machinery, 1986.

[2] S. Kamal Abdali, Guy W. Cherry, and Neil Soiffer. "A Smalltalk system for algebraic manipulation." In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September - 2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 277–283.

[3] Pierre America and Frank van der Linden. "A parallel object-oriented language with inheritance and subtyping." In *Proceedings of OOPSLA '90 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Ottawa, Canada, 21–25 October 1990. *SIGPLAN Notices*, Vol. 25, No. 10, October 1990, pp. 161–168.

[4] Gerald Baumgartner and Vincent F. Russo. *Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism.* To appear in *Software: Practice & Experience*, 1994.

[5] Gerald Baumgartner and Ryan D. Stansifer. *A Proposal to Study Type Systems for Computer Algebra.* RISC-Linz Report 90-87.0, Research Institute for Symbolic Computation, Linz, Austria, March 1990.

[6] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. "Object structure in the Emerald system." In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September - 2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 78–86.

[7] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. "Interfaces for strongly-typed object-oriented programming." In *Proceedings of OOPSLA '89 Conference on Object-Oriented Programming Systems, Languages, and Applications*, New Orleans, Louisiana, 1–6 October 1989. *SIGPLAN Notices*, Vol. 24, No. 10, October 1989, pp. 457–467.

[8] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. "Modula-3 Language Definition." *ACM SIGPLAN Notices*, Vol. 27, No. 8, August 1992, pp. 15–43.

[9] William R. Cook. "Interfaces and specifications for the Smalltalk-80 collection classes." In *Proceedings of OOPSLA '92 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, Canada, 18–22 October 1992. *SIGPLAN Notices*, Vol. 27, No. 10, October 1992, pp. 1–15.

[10] William R. Cook, Walter L. Hill, and Peter S. Canning. "Inheritance is not subtyping." In *Proceedings of 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, 17–19 January 1990, pp. 125–135.

[11] James Donahue and Alan Demers. "Data types are values." *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 426–445.

[12] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Reading, Massachusetts: Addison-Wesley, 1990.

[13] J.A. Goguen and T. Winkler. *Introducing OBJ3.* Technical Report CSL-88-9, SRI International, 1988.

[14] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Reading, Massachusetts: Addison-Wesley, 1983.

[15] Elana D. Granston and Vincent F. Russo. "Signature-based polymorphism for C++." In *Proceedings of USENIX C++ Technical Conference*, Washington, D.C., 1991.

[16] Paul Hudak et al. "Report on the programming Language Haskell: A non-strict, purely functional language, version 1.2." *ACM SIGPLAN Notices*, Vol. 27, No. 5, May 1992, Section R.

[17] Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System.* New York: Springer Verlag, 1992.

[18] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. "An exemplar based Smalltalk." In *Proceedings of OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September - 2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 322–330.

[19] David B. MacQueen. "Modules for Standard ML." *Polymorphism*, Vol. 2, No. 2, 1985.

[20] David B. MacQueen. "An implementation of Standard ML modules." In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, 25–27 July 1988. Association for Computing Machinery, pp. 212–223.

[21] Craig Schaffert et al. "An introduction to Trellis/Owl." In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September–2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 9–16.

[22] Alan Snyder. "Encapsulation and inheritance in object-oriented programming languages." In *Proceedings of OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, 29 September - 2 October 1986. *SIGPLAN Notices*, Vol. 21, No. 11, November, pp. 38–45.

[23] Richard M. Stallman. *Using and Porting GNU CC.* Cambridge, Massachusetts: Free Software Foundation, V. 2.3, 16 December 1992.

[24] Robert S. Sutor and Richard D. Jenks. "The type inference and coercion facilities in the Scratchpad II interpreter." In *Proceedings of SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, 24–26 June 1987, St. Paul, Minnesota. *SIGPLAN Notices*, Vol. 22, No. 7, 1987, pp. 56–63.

[25] Stephen M. Watt, Richard D. Jenks, Robert S. Sutor, and Barry M. Trager. "The Scratchpad II Type System: Domains and Subdomains." In Alfonso M. Miola (ed.): *Computing Tools for Scientific Problem Solving.* London: Academic Press, 1990, pp. 63–82.

[26] Niklaus Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science. Berlin-Heidelberg, Germany: Springer Verlag, 1985.

Base-Class Composition with Multiple Derivation and Virtual Bases

Lee R. Nackman
lrn@watson.ibm.com

John J. Barton
jjb@watson.ibm.com

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598

Abstract

For systems of C++ classes using virtual functions, writing base classes that only specify virtual member functions and then writing other base classes that only implement those functions improves extensibility. When interface is separated from implementation, both interfaces and implementations can be extended separately by derivation. New classes can then be composed by multiple derivation, combining one interface and one implementation base class. We call this form of composition *base-class composition*.

Programmers familiar with the advantages of base-class composition fail to use it because of the performance penalty of multiple derivation and virtual base classes. Consequently, compiler writers, failing to see extensive applications of multiple derivation and virtual bases, have little incentive to eliminate the performance penalty. We highlight the advantages of the base-class composition pattern and show how the performance penalty can be eliminated by compiler optimization.

1 Introduction

In general, C++ classes combine interface and implementation, specifying both the member functions that can be called (interface) and object state (implementation). Derived classes can extend their base class's interface or they can reuse their base class's implementation or both. Building on the work of Martin [1], we observe that there are many advantages to writing base classes that either specify interface or provide implementation, but not both. With a few simple rules applied consistently, these two kinds of cooperating base classes become building blocks that can be combined using multiple derivation.

We call this design strategy *base-class composition*. It allows independent extension of interface, alternative implementation of interface, reuse of implementation, encapsulation of implementation, and avoids recompilation of clients when implementation is altered. Explaining its uses and its advantages are the first goal of this paper.

We believe that programmers familiar with the advantages of this composition fail to use it because of the performance penalty of multiple derivation and virtual base classes. Consequently, compiler writers, failing to see extensive applications of multiple derivation and virtual bases, have little incentive to eliminate the performance penalty. After we highlight the advantages of the base-class composition pattern, we show how the performance penalty can be eliminated by compiler optimization.

We shall focus our attention on aiding the construction of classes to be used through virtual function calls. A function that uses a class is a client of that class; it uses some services provided by the class. Some functions use an instance of a class only through its virtual functions and have no

need to know the exact type of the object being used. Such functions can be written to use pointers or references to a base class with virtual functions. Let's call such a base class an *interface base class*. For example, a function that calls virtual `Shape::draw()` on a `Shape&` uses the object through an interface base class `Shape`. The object itself could be a `Circle`, `Square`, or `Triangle`. Let's call such a function a *use client*.

Base-class composition aids the construction and maintenance of classes for use clients. Two other important clients are *creation clients*, those that create objects (requiring specific types) and *downcast clients*, those that apply derived-type specific operations to objects given only references to common base classes of the objects. Base-class composition does not make creation clients more difficult to design nor does it make downcast clients less difficult.

The next section gives an example of base-class composition. Section 3 discusses characteristics of base-class composition and Section 4 compares it to the alternatives. The performance penalties of using base-class composition are described in Section 5. We show in Section 6 that a straightforward optimization technique can eliminate the costs of multiple derivation with virtual bases under certain circumstances and then, in Section 7, we discuss a further optimization applicable to private bases used in base-class composition. Related work is discussed in Section 8.

2 An Example of Base-Class Composition

To illustrate base-class composition, we use a simple but realistic example. Suppose we want to write a tool that manipulates C++ source code and that we need a way to represent C++ language elements that appear in the source code. We might define a class for each C++ language element to be represented. In particular, let's assume that we want to define classes to represent C++'s `union`, `struct`, and `class`. These are all *aggregates* in the sense that they contain source code entities such as member functions, member data, nested classes, and nested typedefs. However, only `class` and `struct` can have base classes.

It is reasonable to expect classes for representing C++ aggregates to meet the following criteria:

- All three kinds of aggregates should provide common functions representative of aggregation.
- The `class` and `struct` aggregates should provide common functions representative of aggregates that can have bases.
- All three aggregates should be able to use a common implementation of aggregation.
- The two aggregates that can have bases should be able to use a common implementation of access to base classes.

To focus on the issues related to separation of interface and implementation, we limit our example function interfaces: all aggregates respond to `numMembers()` and aggregates that can have bases respond additionally to `numBases()`. We mean for these functions to be representative of larger commonalities and differences between aggregates and aggregates with bases.

Figure 1 shows a class DAG meeting our design goals. Two classes, `Aggregate` and `AggregateWithBases`, are interface base classes, meaning that we intend for client functions to use pointers or references to these classes and that we intend to derive from these classes to implement the interfaces they specify. These classes specify member functions common to their derived classes. For base-class composition we restrict interface base classes to be abstract base classes with neither member data nor constructors.

Characteristics of all aggregates are specified by pure virtual member functions of `Aggregate`:

```
class ostream;
class Aggregate {
public:
  virtual int numMembers() const = 0;
  virtual void kind(ostream&) const = 0; // name of kind of aggregate.
  // ...
};
```

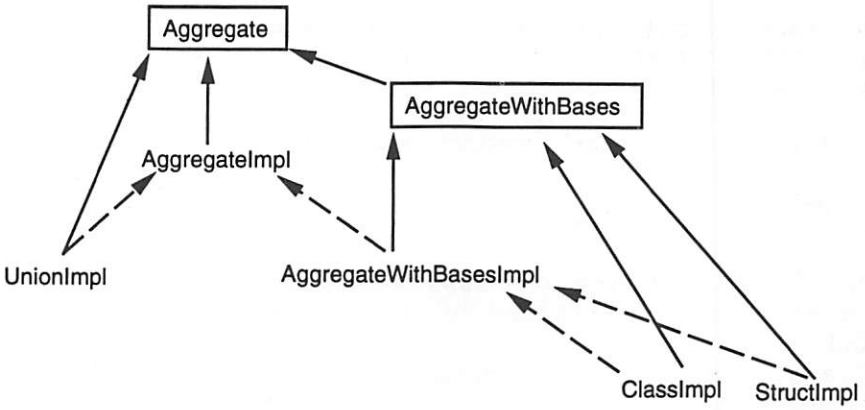


Figure 1: Class DAG for C++ aggregates using base-class composition. Boxes indicate interface base classes. Solid and dashed arrows indicate public and private derivation respectively.

Aggregates that may have bases have all the characteristics of aggregates plus characteristics related to bases. This is specified by publicly deriving `AggregateWithBases` from `Aggregate` and declaring additional pure virtual member functions:

```
class AggregateWithBases :
  public virtual Aggregate {
public:
  virtual int numBases() const = 0;
  // ...
};
```

The reason for using virtual derivation is explained below.
`AggregateImpl` implements the `numMembers()` virtual function of `Aggregate`:

```
class AggregateImpl :
  public virtual Aggregate {
public:
  AggregateImpl(int num_members) : _num_members(num_members) {}
  virtual int numMembers() const { return _num_members; }
  // ...
private:
  int _num_members;  // ...representation of members...
};
```

A `UnionImpl` would represent `union` objects by deriving from this class for code and data reuse:

```
class UnionImpl :
  public virtual Aggregate,
  private AggregateImpl {
public:
  UnionImpl(int num_members) : AggregateImpl(num_members) {}
  virtual void kind(ostream& os) const { os << "union"; }
};
```

`AggregateWithBasesImpl` reuses the implementation of the `Aggregate` layer of its `AggregateWithBases` interface by deriving privately from `AggregateImpl`. It also adds implementation for the `numBases()` virtual function specified by `AggregateWithBases`:

```
class AggregateWithBasesImpl :
  public virtual AggregateWithBases,
  private AggregateImpl {
public:
```

```
  AggregateWithBasesImpl(int num_members, int num_bases) :
    AggregateImpl(num_members),
   _num_bases(num_bases) {
  }
  virtual int numBases() const { return _num_bases; }
private:
  int _num_bases;
};
```

Using multiple derivation here enables `AggregateWithBasesImpl` to implement the `AggregateWithBases` interface on the one hand and to reuse the `AggregateImpl` implementation on the other hand.

Finally, `ClassImpl` is derived from `AggregateWithBasesImpl` using the same style:

```
class ClassImpl :
 public virtual AggregateWithBases,
 private AggregateWithBasesImpl {
public:
  ClassImpl(int num_members, int num_bases) :
     AggregateWithBasesImpl(num_members, num_bases) {
  }
  virtual void kind(ostream& os) const { os << "class"; }
  // ...
};
```

`StructImpl` would be similar.

With these classes we can build and process collections of aggregates. For example, we could construct an array of `Aggregate` pointers that point to `UnionImpl`, `ClassImpl`, and `StructImpl` objects. Then a client function that computes, say, the average number of members for each aggregate would look like this:

```
float avgNumMembers(int n_aggs, Aggregate* aggs[]) {
  int sum = 0;
  for (int i = 0; i < n_aggs; i++) sum += aggs[i]->numMembers();
  return sum / n_aggs;
}
```

Likewise, we can build and process collections of aggregates with bases by creating an array of `AggregateWithBases` pointers that point to `ClassImpl` and `StructImpl` instances. We could then write a client function that asks both for the number of elements as an aggregate and for the number of base classes, say a function that computes the average number of members for classes and structs that don't actually have bases:

```
float avgNumMembersInRoots(int n_aggs, AggregateWithBases* aggs[]) {
  int sum = 0;
  for (int i = 0; i < n_aggs; i++) {
    if (aggs[i]->numBases() == 0) sum += aggs[i]->numMembers();
  }
  return sum / n_aggs;
}
```

This example illustrates how the interface for `AggregateWithBases` layers on top of the interface for `Aggregate`, expressing the idea that aggregates with bases are aggregates. These use clients don't require `ClassImpl` objects to be `AggregateImpl` objects: the implementation is completely hidden from use clients.

3 Characteristics Of Base-Class Composition

Our example illustrates base-class composition. First notice that our example has two kinds of base classes. The interface base classes, `Aggregate` and `AggregateWithBases`, are base classes with

virtual functions, but they are not general C++ classes. They have no implementation. The other kind of base class, `AggregateImpl` and `AggregateWithBasesImpl` could be called *implementation base* classes. They do not add specifications of virtual functions and they are not used as public base classes. Thus our classes separate interface and implementation.

Next notice the relationship between these base classes. The `AggregateImpl` implementation base class derives from the `Aggregate` interface base class, extending it but only adding implementation. The `AggregateWithBases` interface base class also derives from the `Aggregate` interface base class, extending it but only adding more interface specification. The extended interface is implemented in `AggregrateWithBasesImpl` by composing the extended interface with an implementation of the original interface (`AggregateImpl`) and adding implementation for the extended interface. This pattern of implementation, extension, and composition can continue to arbitrary depth; we call the pattern base-class composition.

The C++ language features of `virtual` functions, multiple derivation, `virtual` bases, and the dominance rule for name lookup in the class DAG [2] combine to enable base-class composition. Virtual functions, of course, make the notion of interface possible. Multiple derivation enables an implementation class to derive from both its interface and an implementation base. The combination of virtual bases and dominance connect the interface and the implementation.

Referring to Figure 1, we see a diamond-shaped pattern rooted at `Aggregate`. It has an interface base at the top of the diamond, with the interface extended on one leg (`AggregateWithBases`) and an implementation of the interface base on the other leg (`AggregateImpl`). The class at the bottom of the diamond (`AggregateWithBasesImpl`) completes the implementation of the extended interface specified on one path, building on the implementation along the other path.

`AggregateWithBasesImpl` inherits names from its direct base classes, `AggregateWithBases` and `AggregateImpl`, and it inherits the names of the indirect base class, `Aggregate`. Since `AggregateImpl` and `AggregateWithBases` are derived virtually from `Aggregate`, the name `numMembers` from `AggregateImpl` dominates the same name inherited along the DAG path through `AggregateWithBases`. The function `numMembers()` declared in the public interface base class `Aggregate` is implemented in `AggregateWithBasesImpl` by the `AggregateImpl` base class.

A degenerate triangular version of the diamond-shaped pattern also appears three times in the DAG of Figure 1. The degenerate version omits the extension of the interface. Two of the triangular patterns in Figure 1 are rooted at `AggregateWithBases`, with a partial implementation of the interface provided on one leg by `AggregateWithBases`, and the implementation completed on the other leg by `ClassImpl` (resp., `StructImpl`). Again, multiple derivation, virtual bases, and name dominance combine to yield base-class composition. The third triangular pattern is rooted at `Aggregate`.

Two design conventions—separation of interface and implementation and `virtual` interface base classes—must be adhered to by the programmer to make base-class composition work.

Separation of Interface and Implementation. Base-Class composition relies on the separation of interface and implementation. This means that the specification of the functions callable for an object are separated from the implementation of those functions; for C++, this means that member functions are specified in classes without member data. Classes with member data and implementation of the member functions derive from these classes.

As Martin discussed [1], separating interface and implementation in C++ requires omitting member data in interface base classes and using `virtual` functions. It is also advantageous to declare the functions to be pure virtual so that the compiler will detect attempts to instantiate the interface and detect failure to override base class functions in the derived class. Other languages, including Modula-3 [3] and Ada, provide separate constructs for interfaces and implementations.

Virtual Interface Base Classes. Derivations from interface base classes must be `virtual` if the base-class composition approach is to be applied [1]. Without `virtual` derivation, the interface base classes would be duplicated: `ClassImpl` would have three `Aggregate` interfaces. The

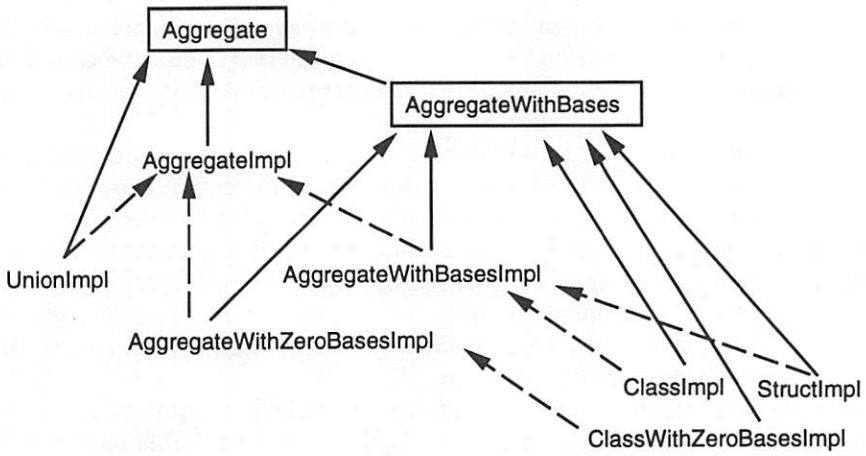


Figure 2: A class DAG extending the DAG in Figure 1 to include additional implementations of the `AggregateWithBases` interface, `AggregateWithZeroBases` and `ClassWithZeroBases`.

implementation names would not dominate the interface names along the `AggregateWithBases` branch and there would be no composition.

These design conventions—separating interface and implementation and deriving virtually from interface bases—represent the programming-time cost of base-class composition. Balanced against these up-front costs are the benefits in maintenance of independent extension, compositions, segregation of use and creation clients, and avoided recompilation. We examine each of these in the following paragraphs.

Independent Extension. Once we have adopted the separation of interface and implementation, the interface can be extended in two ways. Multiple implementations are one kind of extension. For example, we can add `AggregateWithZeroBasesImpl` as shown in Figure 2. This class eliminates the space for the member datum `_num_bases`:

```
class AggregateWithZeroBasesImpl :
  public virtual AggregateWithBases,
  private AggregateImpl {
public:
  AggregateWithZeroBasesImpl(int num_members) :
    AggregateImpl(num_members) {
  }
  virtual int numBases() const { return 0; }
};
```

Of course the saving in this case is trivial because `AggregrateWithBasesImpl` is trivial. Use clients of `AggreateWithBases` work against both implementations.

Adding virtual functions to the interface to create a richer interface is the other kind of extension. For example, we could create a `Class` interface derived from `AggregateWithBases` without affecting the implementation extensions derived from `AggregateWithBases`. These two kinds of extensions are for different purposes, alternative implementation versus additional interface.

Reuse Through Composition. To reuse the `AggregateImpl` implementation of the `Aggregate` interface in the implementation of the `AggregateWithBases` interface layer, `AggregateWithBasesImpl` derives publicly from the interface `AggregateWithBases` and privately from the implementation `AggregateImpl`. Together, these two base classes form a composition that implements the `Aggregate` part of the `AggregateWithBases` interface. The virtual functions defined along the `AggregateImpl` branch (just `numMembers()` in this case) dominate the virtual functions in the `Aggregate` base class along the `AggregateWithBases` branch.

Once we adopt separation of interface and implementation and use `virtual` interface base classes, composition can become a ubiquitous tool in class design. For example, in Figure 2 we can create `ClassWithZeroBasesImpl` simply by composition:

```
class ClassWithZeroBasesImpl :
 public virtual AggregateWithBases,
 private AggregateWithZeroBasesImpl {
public:
  ClassWithZeroBasesImpl(int num_members) :
     AggregateWithZeroBasesImpl(num_members) {
  }
};
```

Here we attach to the extended interface `AggregateWithBases` and reuse one of it implementations. The usage rules for base-class composition are always the same: the interface is public and virtual and the reused implementation is encapsulated with the same consideration as member data, usually as a private base class.

Forcing Access Through Interfaces. In the preceding example code, using private derivation forces member functions to be called through interface references. For example,

```
int totalItems(const ClassImpl& c) {
   // WRONG: AggregateImpl::numMembers() const is a private member
   return c.numBases() + c.numMembers();
}
```

does not compile because public members inherited via private derivation are private. This segregates clients into use clients able to call through the interface only and creation clients unable to call interface functions on objects. This segregation encourages us to write code applicable to all `Aggregate` objects using `Aggregate` references or pointers and to avoid writing code tied to specific implementations of the interface like `ClassImpl`. Whether or not this segregation should be used is a matter of design: some libraries may wish to encourage object access through interface base classes only while others will allow functions at all levels of the class DAG to be called. Access declarations can be used to restore public access in cases in which forcing access through interfaces is not appropriate.

Avoiding Recompilation of Use Clients. Base-class composition also avoids recompilation of use clients when private implementations are redefined, a vital advantage when building large systems. While encapsulation ensures that use clients do not depend on implementation, separating interface and implementation provides a stronger decoupling. Use clients can be compiled to the interface, implementations can be compiled to the interface, and they need only be connected at link time.

4 Why Alternatives are Less Robust

In our experience, base-class composition provides a systematic and robust architectural pattern for object-oriented programs in C++. We have outlined the technique in the preceding section. Here we support our claim that it is more robust—resistant to unforeseen errors—and more maintainable than alternative designs. We pose alternatives in terms of our example.

Combined interface and implementation. We might dispense with the interfaces altogether. A class DAG like that shown in Figure 3 would provide the same implementation reuse as the one in Figure 1. Inheriting interface and implementation together combine four classes into two classes (`AggregateCombined` and `AggregateWithBasesCombined`) and eliminates all multiple derivation. While these may be counted as advantages, client functions are now tied to specific implementations. Modifying the implementation of `AggregateCombined` forces all use clients

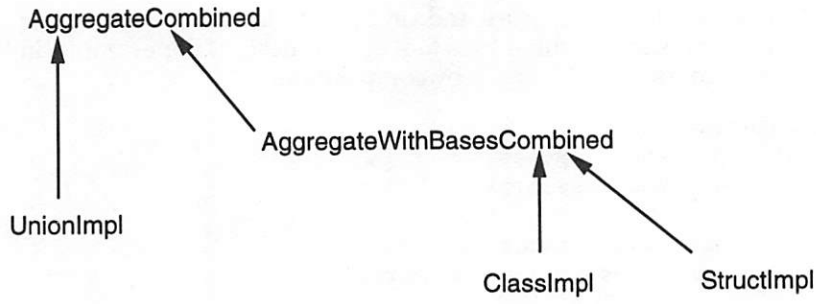


Figure 3: An alternative class DAG design to represent C++ aggregates that uses interface base classes combined with implementation. Compared to Figure 1, the content of `Aggregate` and `AggregateImpl` are combined in `AggregateCombined` and the content of `AggregateWithBases` and `AggregateWithBasesImpl` are combined in `AggregateWithBasesCombined`. Public derivation (solid arrows) must be used here to expose the virtual functions specified in the base classes.

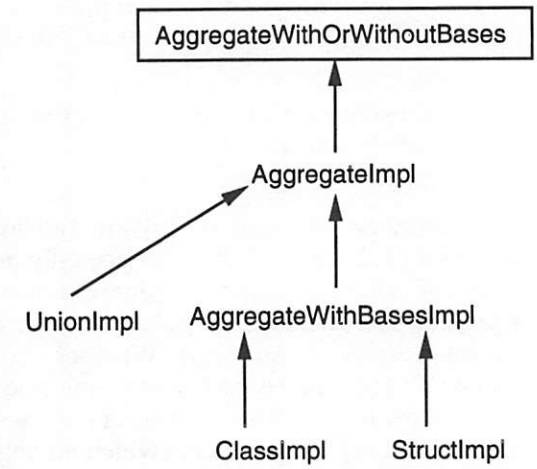


Figure 4: Alternative class DAG using a fat interface in representing C++ aggregates. Compared to Figure 1, this DAG combines the `Aggregate` and `AggregateWithBases` interface base classes into one large interface. Public inheritance must be used here in the derived classes to expose the specifications in the interface.

of `AggregateCombined`, `AggregateWithBasesCombined`, and any further layers to be recompiled when anything is changed.

Combining implementation and interface also prevents alternative implementations from being used by one set of client functions. For example, we cannot define a `ClassWithZeroBasesImpl` object as we did in the preceding section and then use it in use clients expecting `AggregateWithBasesCombined` references or pointers.

We can extend the interface of `AggregateWithBasesCombined` in the same manner that we built it from `AggregateCombined`. For this reason, combined interface and implementation classes work in systems that do not use virtual function interface clients as a major program design element.

Fat interfaces. The number of classes can be reduced by lumping all of the interface functions into one interface base class, say `AggregateWithOrWithoutBases`, as shown in Figure 4. Those parts of the interface not pertinent to a given derived class type are coded in some hopefully harmless way. For example, we could code `AggregateImpl::numBases()` for `AggregateImpl` to always

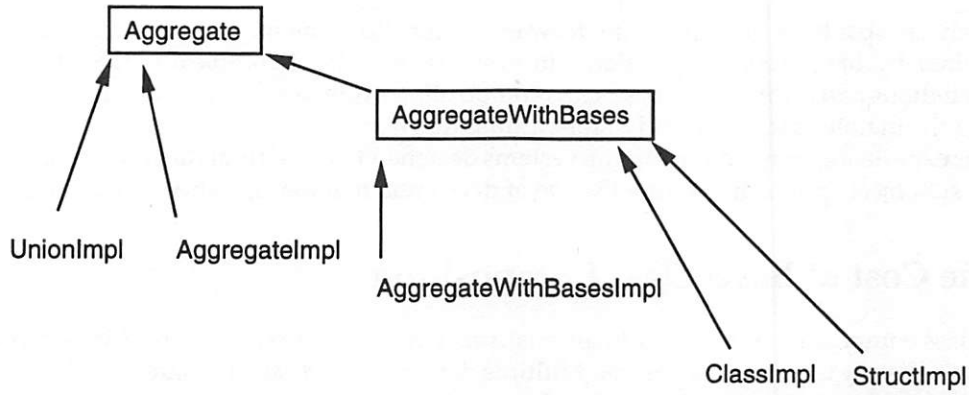


Figure 5: Alternative class DAG representing C++ aggregates using member function forwarding. Compared to the class DAG shown in Figure 1, this DAG has no multiple inheritance. The classes `AggregateImpl` and `AggregateWithBasesImpl` are used as private member data.

return an "impossible" value like `-1` or to throw an exception. Such lumped interfaces, called *fat interfaces* in [4, § 13.6], replace static type checking with either runtime checks or undetected errors.

In addition, the fat interface approach also makes us choose between implementation inheritance and encapsulation. In Figure 4, `ClassImpl` must be derived publicly from `AggregateImpl` to expose the `AggregateWithOrWithOutBases` interface. If we derive `ClassImpl` from the interface publicly and directly we cannot also inherit the implementation of `AggregateImpl` privately unless we build a structure equivalent to the base-class composition. If we extend the interface, adding functions to `AggregateWithOrWithoutBases`, all use clients must be recompiled, even if they use only the equivalent of the `Aggregate` interface.

We can add new implementations in the fat interface approach and these implementations don't require recompilation of use clients. For this reason, fat interfaces appear in small projects with heavy use of virtual functions.

Implementation Reuse via Member Subobjects. As a final alternative, we abandon implementation reuse through inheritance but retain the layered interface, as shown in Figure 5. Instead of obtaining an `AggregateImpl` subobject via inheritance, `AggregateWithBasesImpl` could have a member subobject, like this:

```
class AggregateWithBasesImpl :
  public AggregateWithBases {
public:
  AggregateWithBasesImpl(int num_members, int num_bases) :
    _aggregate_impl(num_members),
   _num_bases(num_bases) {
  }
  virtual int numMembers() const { return _aggregate_impl.numMembers(); }
  virtual int numBases()   const { return _num_bases;           }
private:
  AggregateImpl _aggregate_impl;
  int _num_bases;
};
```

The function `numMembers()` is said to be *forwarded* to the member `_AggregateImpl`. This seems fine until you try to compile the class and get an error message. `AggregateImpl` is an abstract base class since it doesn't implement the pure virtual function `kind()`. To use forwarding in this situation, you must add a dummy implementation of `kind()` to `AggregateImpl`.

Once the code is correct, this alternative is indistinguishable from the base-class composition as far as client functions are concerned: the interfaces are identical and the implementations are

completely encapsulated. However, the forwarding functions themselves are a maintenance item not required by base-class composition. In systems of realistic complexity, this style of reuse becomes tedious and error-prone. Base-class composition expresses the same relationships without imposing the maintenance overhead of forwarding functions.

In our experience, larger more mature systems designed to use virtual functions often adopt the member subobject approach, limiting the use of derivation to building subtype relations.

5 The Cost of Base-Class Composition

If base-class composition is superior to alternatives, why isn't it used in more object-oriented C++ programs? History is part of the answer. Multiple derivation, virtual bases, and dominance lookup are relatively new additions to C++ and early experiences with these features were not altogether positive (see Section 8). But even those programmers aware of the advantages select alternatives as a practical engineering tradeoff. The problem is performance.

As far as we are aware, current C++ compilers use roughly the scheme outlined in [2] to implement virtual bases with virtual functions. Minor optimizations aside, this scheme adds to each object one virtual function table pointer per base class and creates one virtual function table per base class for each class. In addition, one pointer to each virtual base class is needed for each subobject declaring a virtual base class. Thus we would expect `ClassImpl` in Figure 1 to take 11 words of memory: two words for the integer members, five words for virtual function table pointers, and four words for pointers to virtual bases. (The compiler we tried this on used ten words, saving a word by sharing two virtual function table pointers.) The size overhead carries a proportional runtime overhead since each pointer in the object must be initialized when the object is created.

Since each layer of base-class composition adds at least two base classes with virtual functions, these overheads increase with increasing layers. Such scaling works against the application of base class composition to the very kinds of problems—large systems—at which the technique is most adept.

6 Optimization of Base-Class Composition

The costs of base-class composition can be eliminated by a straightforward compiler optimization. This claim would be best proven by a C++ compiler that did not have the overheads of existing implementations. To convince compiler writers to implement such an optimization, we need both the motivation that we presented in the preceding sections and convincing arguments that the optimization will succeed. In this section we outline the optimization and argue that it will succeed.

The core of our argument lies in recognizing that the restricted interface base classes we need for base-class composition only specify the contents of virtual function tables, contents that are determined at compile time. Thus we can transform the class DAG at compile time to eliminate the pointers needed for more general virtual base classes, as long as the virtual function tables are filled correctly.

We begin with a definition: a *pure abstract base class* is an abstract base class (a class having at least one pure virtual function) with no data members and only C++-generated constructors, possibly derived from other pure abstract base classes. This restricted kind of interface base class is the kind we advocate for base-class composition. A compiler can test for a pure abstract base class unambiguously by examining a class and its base classes.

We claim that a DAG containing pure abstract base classes can be transformed into an equivalent DAG having no virtual base class derivation involving a pure abstract base class. By equivalent we mean that programmers will not be able to detect the difference by ordinary means—more on that shortly. Obviously, if this claim is true, no virtual base pointers are needed since there are no virtual bases.

Removing virtual derivation "unfolds" the pure abstract base class portion of the DAG into a tree, replicating some of the pure abstract base classes. Each resulting tree branch can be implemented

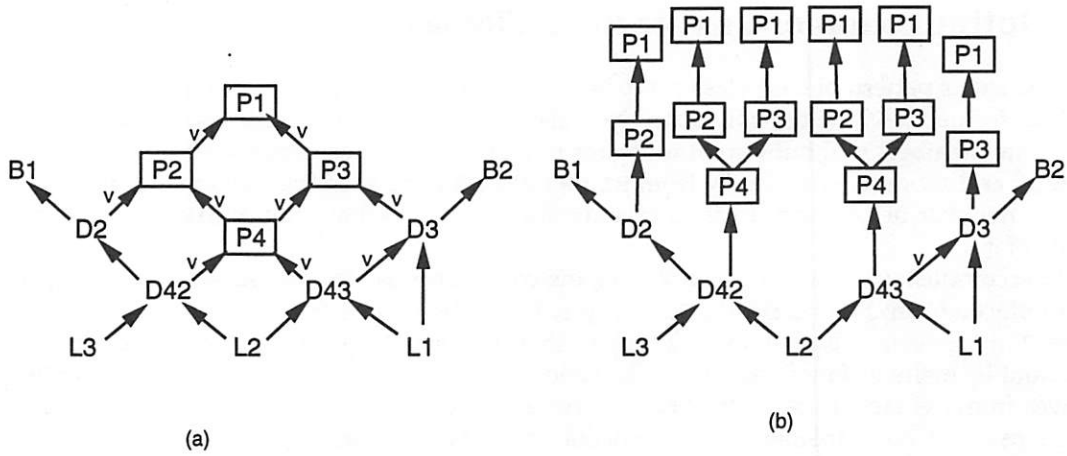


Figure 6: A hypothetical class DAG, (a), and the result of optimizing it (b).

with a single virtual function pointer [2]. Consequently, the transformation eliminates virtual base pointers and most of the pointers added by multiple derivation.

To understand this transformation, consider the example class DAG in Figure 6a. Classes `P1`, `P2`, `P3`, and `P4` are pure abstract base classes; the other classes are arbitrary. The class names starting with `D` border on the part of the DAG that will be transformed; those starting with `B` and `L` are just other parts of the DAG. Arrows point to base classes and virtual derivations are marked with a `v`. For example, the derivation of `D43` from `D3` is virtual but `D3` is not a pure abstract base class.

The transformed DAG is shown in Figure 6b. With the virtual derivations from pure abstract base classes changed to non-virtual derivations, the pure abstract base classes are duplicated in the resulting DAG. For the present DAG, base `P1` appears 6 times as base classes for type `L2` rather than once. However, since none of the classes that are duplicated contain data, the storage required for any object from any class in the transformed DAG is less than that of classes in the original DAG.

Now we claim that these DAGs are equivalent in the following sense: the source code using the original DAG can be rewritten to use the transformed DAG such that the rewritten code works the same as the original in all ways that do not depend on implementation dependent object-layout details. Object sizes, offsets of members, and the structure of virtual function tables will change (good!), but otherwise the program will be the same.

After the DAG transformation, the compiler must (a) forward member function calls defined along one replicated path to the other path, and (b) disambiguate pointer and reference conversions from derived classes to replicated pure abstract base classes. For example, member functions in `D2` that override pure virtual functions in `P1` or `P2` will have to be overridden in `D42`. `P1` pointers initialized with `D42` pointers will have to be converted up through one path, say `P4` to `P2` to `P1`. All paths are equivalent since `P1` is replicated by design.

The rewriting is sufficient because there are only two things one can do with a pure abstract base class: (1) initialize a reference or pointer of the base type with a reference or pointer, respectively, of one of its derived types, or (2) call one of its member functions through a reference or pointer. The initializations can be done with replicated classes caused by removing the virtual specifiers just as well as to the shared classes because the contents of the replicated classes are fixed at compile time. Calls to member functions give the same result for both shared and replicated classes because the functions have the same definitions.

Programmers using objects from classes derived from pure abstract base classes cannot detect whether or not these objects contain shared or replicated subobjects from these bases. Again, the lack of member data in pure abstract base classes allows the optimization: there is no shared data to point to or initialize. In Figure 6b, class `D43` derives virtually from `D3` even in the transformed DAG because `D3` is not a pure abstract base class and cannot be optimized in the manner described here.

7 Optimization of Private Base Classes

The particular pattern of base-class composition can be further optimized: Private base classes in the transformed DAG whose only public base classes are pure abstract base classes can be rewritten as private members without virtual functions of their own. For example, if `D42` derived privately from `D2` and `D2` derived privately from `B1` then the `D2` portion of the DAG would be stored as a private member of `D42` and, further, this member would not have any virtual function pointers inside of it.

To accomplish this second optimization, the compiler must rewrite classes into a concrete and an interfaced form. For a class like `D2` in Figure 6, call the type of the concrete form `ConcreteD2`. Class `ConcreteD2` is exactly like `D2`, except that it does not derive from any pure abstract base class and its member functions are not declared virtual. A new interfaced `D2` class is written that derives from the same pure abstract base classes of the original `D2` and declares the same member functions as `D2`, but implements these functions by forwarding to a private `ConcreteD2` member datum.

All clients of the interfaced `D2` see the same behavior as they saw from the original `D2`. However, the compiler now has a private representation class, `ConcreteD2`, that does not have virtual functions and hence has no virtual function pointers. This private representation can be used whenever `D2` appears as a private subobject, either as a private member datum or, as in the case of base-class composition, as a private base class.

This second optimization succeeds because of encapsulation and because, unlike base-class pointers or references that can bind to different objects types at run-time, member objects have a fixed known type. As the compiler implements a private subobject, it has all of the source code that can call member functions on that subobject (encapsulation) and it can resolve virtual function calls on that subobject at compile time without going through a virtual function table (one type). Thus private subobjects can be implemented using the concrete private representation of a class.

To illustrate the impact of these optimizations on a recognizable example, we return to the C++ aggregate example. Here is part of the result of applying these optimizations by hand:

```
class ConcreteAggregateImpl {
public:
  ConcreteAggregateImpl(int num_members) : _num_members(num_members) {}
  int numMembers() const { return _num_members; }
private:
  int _num_members;
};

class AggregateImpl :
  public Aggregate {
public:
  AggregateImpl(int num_members) : _aggregate_impl(num_members) {}
  virtual int numMembers() const { return _aggregate_impl.numMembers(); }
private:
  ConcreteAggregateImpl _aggregate_impl;
};

class ConcreteAggregateWithBasesImpl {
public:
  ConcreteAggregateWithBasesImpl(int num_members, int num_bases) :
    _aggregate_impl(num_members),
    _num_bases(num_bases) {
  }
  int numBases() const { return _num_bases; }
  int numMembers() const { return _aggregate_impl.numMembers(); }
private:
  ConcreteAggregateImpl _aggregate_impl;
  int _num_bases;
};
```

```
class AggregateWithBasesImpl :
  public AggregateWithBases {
public:
  AggregateWithBasesImpl(int num_members, int num_bases) :
    _agg(num_members, num_bases) {
  }
  virtual int numBases() const { return _agg.numBases(); }
  virtual int numMembers() const { return _agg.numMembers(); }
private:
  ConcreteAggregateWithBasesImpl _agg;
};
```

The concrete version of each class provides pure implementation, without any connection to the rest of the class DAG. As a result, there is no overhead from virtual base pointers or virtual function pointers. These concrete classes provide implementation reuse. Since each non-interface class is derived through a single inheritance chain, only one virtual function table pointer is needed. Thus, the overhead of this scheme is one word: an instance of the transformed `ClassImpl` takes three words, two for the integer member data and one for a virtual function table pointer.

Separation of interface and implementation is central to the success of the transformation. Imagine for the moment that our `Aggregate` base class had some member datum. As a virtual base class, a single copy of this datum would be included in a `ClassImpl` object. But in transforming this system to single-inheritance, a private `AggregateImpl` member datum would replace the private base class and two copies of the `Aggregate` member would be enclosed in a `ClassImpl` object: one from the `AggregateImpl` member and one remaining in the `Aggregate` base class. The pointers introduced by C++ compilers to support virtual function calls in the presence of multiple inheritance and virtual bases give offsets to adjust the `this` pointer to accommodate base class member data. Separation gives interfaces with no data; no data means no pointer offsets.

8 Related Work

Separation of Interface and Implementation. Section 3 follows the work of Martin [1]. He started with an "ideal" model for interface-based programming and showed that a C++ programming style could almost be equivalent. He concluded with a suggestion that would improve the language support for this kind of style. Since Martin's paper appeared, the C++ language has been altered and the technique we call base class composition is the same as the style advocated by Martin but using the new language rules. As Martin predicted, the result is a clean mechanism for programming with interfaces, as we illustrate here.

Martin's assessment of the performance penalty omitted runtime initialization of the virtual-base and virtual function table pointers. In our experience, C++ programmers prefer the cost of maintaining forwarding functions and even live with mixed interface and implementation rather than accept the space and time overheads. For this reason, we investigated compiler optimization for this important technique.

We have also distinguished the kinds of program designs—those that rely on use clients—that will benefit from separation of interface and implementation. Programs that do not use virtual function calls will only see the programmer-time cost of base class composition and no benefit.

We Need Multiple Derivation. Cargill has argued [5, 6] that many cases of multiple derivation can be reimplemented using single derivation with some advantages and that, given the complex semantics of multiple derivation, it is not a useful language feature. In [6], he gives three examples using multiple inheritance. The two he argues are failures do not separate interface and implementation; moreover the implementations are not layered to allow reuse. His third example, the one he thinks may be a reasonable use of multiple inheritance, uses two interfaces but no implementation reuse.

We believe Cargill's conclusion was reached without the benefit of enough examples of large systems of classes designed for extensibility. Waldo[7] analyzed Cargill's arguments against multiple derivation by suggesting that three kinds of base classes might be distinguished: 1) those for implementation inheritance, 2) those for interface inheritance, and 3) those for data inheritance. He argued that Cargill focused on the first, but that the latter two are more important in large systems and that multiple derivation is needed to support the use of such base classes.

Waldo's classes for implementation inheritance correspond to our classes with combined interface and implementation. We agree with his arguments for interface inheritance, but we see a broader role for its use. By splitting combined interface and implementation classes then composing these classes via multiple derivation we achieve the benefits Waldo cites for interface inheritance and we achieve implementation reuse. Base-class composition provides simple and consistent guidelines allowing us to ignore the complexity of the general multiple derivation in C++.

We Need The Dreaded Virtual Base Diamond. Meyers [8] also argues against multiple derivation and especially against virtual bases used in the "dreaded diamond-shaped inheritance graph" (p. 165). He argues that one cannot predict when a virtual base should be used; we say that every derivation from an interface base should be virtual. He argues that constructors for virtual bases are problematic; we say don't put data in abstract base classes and you won't need constructors in virtual bases. He argues that ambiguities can arise in multiple derivation and that dominance is mysterious; we say that using multiple derivation in a disciplined way with one branch adding new virtual functions and the other implementing previous virtual functions harnesses the mysterious powers of the dominance rule. He argues that virtual bases do not allow casting a pointer to a base class to a pointer to a derived class; we agree with his assessment that this is not very important.

Meyers then goes on to give an example of multiple derivation using public interface and private implementation base classes. He sees this example as useful and comprehensible, but says (p. 165) that "it's no accident that the dreaded diamond-shaped inheritance graph is conspicuously absent." However, if he simply added virtual derivation to his graph, he could have completely eliminated the member function definitions in his final class, together with their maintenance.

Meyers, like Cargill, exposes many problems with the use of multiple derivation and the use of virtual bases. Their arguments focus on practical examples and they admit possible exceptions. We agree that the uses they explore do have problems. However, we blame failure to separate interface and implementation for most of the problems and find important uses for both multiple derivation and virtual bases used for base-class composition.

Cargill [6] and Meyers [8] advocate manual rewriting as means of avoiding the overhead of multiple derivation and virtual base classes. In effect they favor a coding style that replaces base-class composition by hand optimization. Here we have demonstrated that base-class composition is simpler and easier to maintain; it can be a key building block for layered systems and the optimization can be done automatically.

Not All Public Base Classes Should be Virtual. Sakkinen[9] discusses the C++ inheritance model and concludes that public base classes should always be virtual while private base classes should always be nonvirtual. Base class composition does conform to these guidelines, but we disagree with the guidelines in general. Without virtual functions, public base classes have a very different role in class designs and we cannot decide to make them virtual in all cases. With mixtures of interface and implementation, we must let the required object state dictate the choice of virtual or nonvirtual bases.

In another paper,[10] Sakkinen discusses initialization problems with virtual base classes. Again, base class composition uses no constructors in virtual bases and initialization is therefore not relevant to the programmer.

9 Conclusion

Base-class composition allows large, layered systems of interfaces to be implemented robustly and simply. Separate interface and implementation base classes are composed to form a base for further implementation without compromising extensibility or encapsulation. We believe that when virtual functions are used by client functions—when programming through interfaces—base class composition should become an important tool for C++ programmers. However, the performance barrier must be overcome first. We have proposed optimizations that allow compilers to eliminate all but one of the pointers needed to implement C++ for systems using base class composition.

Acknowlegements

We appreciate the suggestions for improving this paper that were made by Michael Karasick, Derek Lieber, Chris Laffra, Barry Rosen, Michael Fraenkel, Noel Sales, Ralph May, Paul Golick, and Lars Hougaard. We also thank Ernie Choi for his encouragment.

References

[1] Bruce Martin. The separation of interface and implementation in C++. In **USENIX C++ Conference Proceedings**, pages 51–63. USENIX Association, April 1991.

[2] Margaret A. Ellis and Bjarne Stroustrup. **The Annotated C++ Reference Manual**. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1990.

[3] Samuel P. Harbison. **Modula-3**. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[4] Bjarne Stroustrup. **The C++ Programming Language**. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, second edition, 1991.

[5] T.A. Cargill. Does C++ really need multiple inheritance. In **USENIX C++ Conference Proceedings**, pages 315–323. USENIX Association, April 1990.

[6] Tom Cargill. **C++ Programming Style**. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

[7] Jim Waldo. Controversy: The case for multiple inheritance in C++. **Computing Systems**, 4(2):157–171, 1991.

[8] Scott Meyers. **Effective C++: 50 Specific Ways to Improve Your Programs and Designs**. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

[9] Markku Sakkinen. A critique of the inheritance principles of C++. **Computing Systems**, 5(1):62–110, 1992.

[10] Markku Sakkinen. How should virtual bases by initialized (and finalized)? **C++ Report**, 5(3):44–50, 1993.

Faster Parsing via Prefix Analysis

Martin D. Carroll
carroll@research.att.com

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974-2070

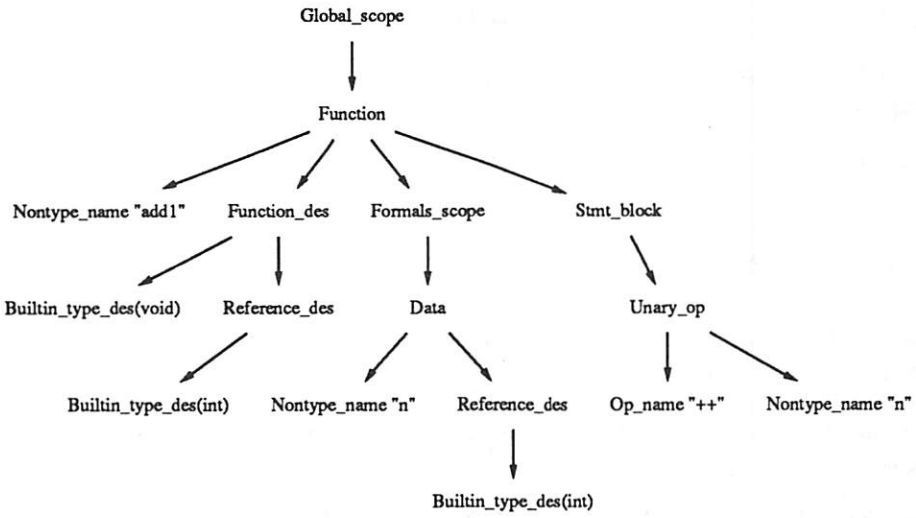
Many C++ programming environments represent C++ code as some form of tree. Producing the tree for a given piece of code is called *parsing*. One important optimization a C++ environment can provide is to avoid repeated parsing of header files. Unfortunately, trying to implement this optimization by using precompiled header files has several drawbacks. This paper presents a new solution based on prefix analysis. Specifically, it shows how to speed up parsing by finding a near-maximal prefix of the given translation unit that was already parsed at some time in the past.

1 The Problem

Many C++ programming environments represent C++ code as some form of tree. For example, C++ compilers typically represent code as a parse tree with respect to a particular grammar. Other environments use trees that are unrelated to any grammar. For example, the ALF tree [1] for the code

```
void add1(int& n) {
    ++n;
}
```

is the following:



This paper will slightly abuse the term "parse tree" to refer to whatever tree is used to represent a piece of code, and the term "parsing" to mean "constructing the parse tree."

In most programming environments, parsing is done often. Hence, we would like it to be as fast as possible. One source of optimization is the fact that often a large prefix of the translation unit currently being parsed was already parsed at some time in the past. For example, consider this file:

f.c:

```
#include <X.h>

void f() {
    // ...
}
```

If we parse this file, then change the definition of `f`, then parse the file again using the same `-D`, `-U`, and `-I` options, and if the header files have not changed, then the prefix of the translation unit corresponding to the macro-expansion of *X.h* will have already been parsed. If we later parse a different file

g.c:

```
#include <X.h>

void g() {
    // ...
}
```

that prefix (again assuming that things have not changed) will again be the same. We would like our parsing algorithm to exploit these common prefixes.

A parser that exploits common prefixes should have the following properties:

- It should be *correct*. It should go without saying that programs should be correct. Unfortunately, many existing implementations of precompiled header files (see Section 2) are not correct, but are used by thousands of C and C++ programmers every day.
- It should be *efficient*. In particular, it should be never be significantly slower than a parser without the optimization, and it should be faster when an already-parsed prefix exists.
- It should be *maximal*. That is, the time saved when parsing a translation unit should be proportional to the time that would have been taken to parse the already-parsed prefix.
- It should be *transparent*. That is, the parser should perform the optimization with no (or perhaps very little) intervention by or assistance from the user.
- It should be *easy to implement*.

2 Precompiled Headers

One solution to the common prefix problem used in many programming environments ([2, 3], for example) is precompiled header files. A *precompiled version* of a header file f is a representation of the code that is the result of macro-expanding f. The macro-expansion of f depends on the `-D`, `-U`, and `-I` options that are given to the preprocessor, as well as the contents of the file system when the preprocessor executes.

Suppose that after someone generates a precompiled version of *X.h*, a programmer parses a file whose first nonwhitespace, noncommentary code is the following `#include` statement:

```
#include <X.h>
// ...
```

If the result of preprocessing *X.h* using the current `-D`, `-U`, and `-I` options and the current contents of the file system would be the same as the macro-expansion corresponding to any stored precompiled version of *X.h*, then the parser can use that precompiled version, and avoid preprocessing and parsing *X.h* again.

Notice that if any nonwhitespace, noncommentary code precedes the `#include`,

```
#define Widget Gigdet
#include <X.h>
// ...
```

then it is in general not safe to use a precompiled version of *X.h*. It is not even necessary to use the preprocessor to render the precompiled versions unsafe to use. For example, consider the following code:

```
int Widget;
#include <X.h>
// ...
```

If the code in *X.h* uses `Widget` only as a type name, then declaring `Widget` as a nontype name before including *X.h* causes a parse error because of the so-called $1\frac{1}{2}$ name space rule [4]. Hence, the precompiled versions of *X.h* are unsafe to use.

The scheme just described is not maximal. If two files share a long initial prefix of `#include` statements, the precompiled version of only the first `#include`d file is used. This problem can be solved by recognizing initial *sequences* of `#include` statements, and storing precompiled versions of `#include` sequences. This strategy is sufficiently difficult to implement that not all implementations of precompiled headers provide it.

The principle drawback of precompiled headers is that they are difficult to implement correctly, efficiently, and transparently. Consider trying to determine whether any of the stored precompiled versions of a header file *f* can be used. Even if the `-D`, `-U`, and `-I` options are the same as those originally used to create a precompiled version of *f*, and even if the values of the `__DATE__`, `__TIME__`, and `__FILE__` preprocessor variables are the same, the contents of the file system might have changed. In particular, if any of the header files directly or indirectly included by *f* has changed, the macro-expansion of *f* will probably be different, in which case the precompiled version of *f* cannot be safely used. Even if the contents of those header files has not changed, if any new files with the same names as included files, but earlier on the `-I` path, have appeared in the file system, the macro-expansion of *f* will probably be different.

Determining whether the contents of the file system has changed in a way that renders a precompiled version of a header file unsafe to use is difficult to implement correctly and efficiently. Hence, many implementations of precompiled headers either are incorrect, or foist part or all of the problem onto the users. For example, some implementations permit only one precompiled version of a header to exist at any time, and assume that the precompiled version of a header can always be safely used. Of course, if this assumption is wrong, the parser has undefined behavior.

Although tools (such as `nmake` [5]) exist that can help users track header file dependencies, implementations of precompiled headers that require users to do *any* such tracking are not transparent. Further, because there are no tools that make tracking header files sufficiently easy, most users of precompiled headers do not completely track their header files. When such users parse their code, they have no assurance that the resulting parse tree is correct.

3 A Better Solution

This paper presents a solution to the common prefix problem that is correct, efficient, near maximal, transparent, and relatively easy to implement.

The key to this solution is to recognize that analyzing code before it has been preprocessed is hopeless. As one anonymous reviewer of this article stated, "The bottom line is that the preprocessor is a law to itself that you cannot trust not to change the program semantics in unpredictable ways, so the only way to reasonably deal with it is to take it out of the picture somehow...." We take it out of the picture by *first* completely preprocessing the code to be parsed, *then* performing prefix analysis.

Our solution requires that the parse trees we are generating have one reasonable property: The top-level statements in the translation unit must correspond to the level-one subtrees in the parse tree. (A *level-one subtree* of a tree T is a tree whose root is a child of the root of T.) For example, the tree for the code

```
class T {
    // ...
};

void add1(int& n) {
    ++n;
}

extern int i;
```

must have three level-one subtrees corresponding to the definition of `T`, the definition of `add1`, and the declaration of `i`.

Here is a high-level description of our parsing algorithm:

Algorithm 1

1. preprocess the code
2. find the maximal prefix π of top-level statements that has already been parsed
3. create a parser in the state that it would have after parsing π
4. parse the remaining text, storing the results

The next section will precisely define the "state" of a parser. Notice that performing the prefix analysis on the code *after* preprocessing completely eliminates the need for users to track header file dependencies. The tradeoff is that the parser must always preprocess all code, even the code for prefixes that have already been parsed. As we shall see in Section 11, a fast preprocessor can reduce the time spent in step (1) to a tolerable level. Notice also that to determine the maximal prefix in step (2) we need a data structure that is stored across executions of the parser. Such a data structure is described in Section 5.

Incidentally, notice that our algorithm has the desirable property that if we parse some code, then change a comment, then parse again, the second parse is fast, because changing a comment does not affect the text of the resulting translation unit. (Preprocessing replaces each comment with a single space.)

4 Parser State

Let s be a sequence of top-level C++ statements. The *parser state after* s is the minimal amount of information needed to create a parser that will correctly parse any code following s. Note that the definition of parser state depends on the kind of trees we are building. For the kind of trees typically built, the parser state is information about the names declared in s. For example, consider the following code:

```
class T {
public:
    void f();
};

void add1(int& n) {
    float g;
    ++n;
}

extern int i;
```

The parser state after this code is typically something like the following:

> Type names `T`, `add1`, and `i` are declared at global scope; nontype name `f` is declared in `T`'s inner scope.

The state does not include anything about the names `n` or `g` because nothing about these names can affect the parse of any code following this sequence.

Now suppose that s is the concatenation of the top-level statements $s_1, s_2, \ldots, s_n$. Then the parser state after s is $(\Delta_n \circ \cdots \circ \Delta_2 \circ \Delta_1)(0)$, where 0 is the initial parser state and the *parser delta* Δ_i represents the effect of s_i on any given parser state. For example, the following is the parser delta for the definition of `T`:

> Add type name `T` to the global scope; add nontype name `f` to `T`'s inner scope.

In practice, we can efficiently create a parser in the state after s by creating a parser in the initial state and applying all the parser deltas.

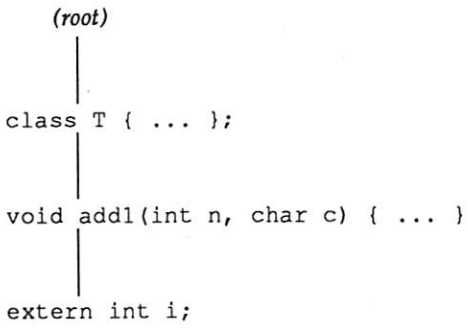
5 The Prefix Tree Data Structure

The data structure that we will use to store information about already-parsed code is a *prefix tree*. To avoid confusion with the trees used to represent C++ code, we will always use the complete term "prefix tree" when referring to this data structure.

Each node other than the root in a prefix tree represents a top-level C++ statement. The path from the root to any node represents the translation unit formed by concatenating the statements represented by the nodes along that path; the path from the root to itself represents the empty translation unit. Each node n in a prefix tree contains the following information:

- TEXT_n — the text of the top-level statement s represented by n
- TREE_n — the level-one subtree that represents s
- Δ_n — the parser delta for s

For example, the following is the prefix tree (showing only the values of TEXT_n) for the code in the previous section:

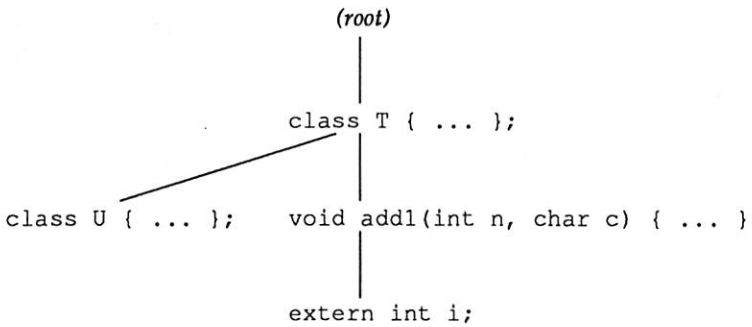


Now consider this translation unit:

```
class T {
public:
    void f();
};

class U {
public:
    void g();
};
```

If we represent this translation unit in the same prefix tree, it looks as follows:



Deciding how many prefix trees to use in a programming environment, and what parses should use what prefix trees, is purely a policy decision of the environment. If we use the same prefix tree for a series of parses, the prefix tree can grow arbitrarily large. Hence, it would potentially be too inefficient to read the entire prefix tree at the start of every parse that uses it. To solve this problem, we use a disk-based prefix tree; that is, we read in only those nodes that we need to examine.

6 The Compression Problem

To minimize the time we spend reading TEXT_n for all the nodes that we will need to examine while parsing, we will store TEXT_n in compressed form.

One way to achieve high compression is to compress TEXT_n using a compressor that is in the state s resulting from having compressed all the ancestors of n, starting from the root. Unfortunately,

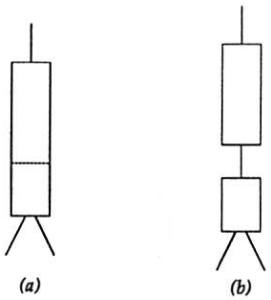


Figure 1: Splitting a node in the prefix tree. (a) Before split. (b) After split.

if we use this strategy, the parsing algorithm is difficult to implement efficiently. In the parsing algorithm we will potentially need to uncompress all the children of a given node. If each child was compressed using a compressor in state s, we will need to set the state of our uncompressor to s for each child. For many compression algorithms, it is not easy to efficiently set the state of an uncompressor to a noninitial state s.

Thus, we decide to compress each TEXT_n individually. Unfortunately, an unscientific survey done by the author suggests that the average size of a top-level statement in well-written C++ code is about 200 characters; compressing each TEXT_n individually would not achieve a high compression rate.

To solve this problem, we will slightly change our definition of prefix tree and let every node in the prefix tree represent a contiguous *sequence* of top-level statements. The most elegant approach is to have every node represent as many top-level statements as is necessary to give the tree the property that no node has only one child. (If a node had only one child, the child could be merged with that node.) When we search for the longest prefix that has already been parsed, if that prefix ends within the sequence of statements represented by a node, we split the node before continuing (see Figure 1).

This elegant approach is difficult to implement efficiently, however. First, when examining a node, we cannot simply read in the entire compressed text stored at that node, because there would be no upper bound on the size of that text. We must instead read in a constant-sized chunk at a time, uncompressing as we go. Further, splitting a node is problematic. Suppose that a node n that we wish to split represents the (uncompressed) text $t = t_1t_2$, and we wish to split n between t_1 c and t_2. Let c be the compression of t. We must compute the compressions c_1 and c_2 of t_1 and t_2, respectively. Computing c_1 is easy: It is simply a prefix of c. The compression c_2, however, is in general *not* the remaining suffix of c. To compute c_2 we might first compute the uncompressed t_2, then compress it. However, computing t_2 requires reading in the entire compressed text stored at n, which, as already stated, we do not wish to do.

To solve these problems, let us instead have every node in the prefix tree represent a contiguous sequence of some constant number k of top-level statements. (We will usually have to settle for fewer than k at any node representing the end of a translation unit.) There exists a fast compression algorithm [6] such that compressing 50 top-level statements achieves in practice a compression of approximately 70%. This rate is approximately that achieved by compressing the entire translation unit at once. Thus, we choose $k = 50$.

Letting each prefix tree node represent up to 50 top-level C++ statements means that our algorithm will not always find the maximal already-parsed prefix in step (2) of Algorithm 1. Instead, it will find a prefix of top-level statements that contains at most 50 statements fewer than the

maximal prefix. However, if we assume a constant upper bound on the numbers of characters in a top-level statement, then our algorithm is short by only a constant factor. Further, representing 50 statements per node reduces several constant factor overheads in our algorithm that we do not bother discussing in this paper.

7 Parsing a Prefix

To construct the nodes in the prefix tree, we need a function that can parse at most 50 top-level statements from the text remaining in a translation unit. We can then call this function repeatedly to parse the entire translation unit.

More specifically, let s be a sequence of top-level statements, let P be a parser in the state after s, and let e be a text string. Suppose that e, when parsed in the context of s, is error free, and let t be the first 50 (or fewer, if there are fewer than 50) statements in e. Then the function *parseprefix*(P, e) returns the triple $\langle len, T, \Delta \rangle$, where *len* is the number of characters in t, T is the parse tree representing t, and Δ is the parser delta for t. Further, when *parseprefix*(P, e) returns, P is in the state after $s \cdot t$.

In the remainder of this paper, we will assume that e, when parsed in the context of s, is error free. Removing this assumption requires only that we add a few uninteresting complications to the algorithms we present.

If a parser is implemented cleanly, implementing the function *parseprefix* is relatively easy.

8 The Algorithm

Here is our parsing algorithm (Algorithm 1) in greater detail:

Algorithm 2

parse(t, T) is

```
1     e ← macro expansion of t
2     P ← parser in initial state
3     n ← root of T
4     while e is not the empty string
5         if there is a child m of n such that TEXT_m is a prefix of e
6             apply Δ_m to P
7             remove the prefix TEXT_m from e
8             n ← m
9         else
10            ⟨len, T, Δ⟩ ← parseprefix(P, e)
11            create a new node m = ⟨e_0..len-1, T, Δ⟩
12            make m a child of n
13            remove the prefix TEXT_m from e
14            n ← m
15    return n
```

First we macro expand the translation unit, and we create a parser in the initial state. Then beginning at the root of the prefix tree T we walk down the tree. At each node we look for a child whose text matches a prefix of the text e remaining in the translation unit. If there exists such a child, we move to it, strip that prefix from the remaining text, and apply the parser delta to the parser. Otherwise, we call *parseprefix* to parse a prefix of the remaining text; then we create a prefix tree node with the information returned by *parseprefix*, make it a child of the current node, and move to that child. We are finished when there is no more text remaining in the translation unit.

As mentioned in Section 5, we should read from disk only those nodes of the prefix tree that we must examine, which in this algorithm occurs on line 5. Also, we should free the memory used to hold a node as soon as we leave that node on lines 8 and 14 (but see Section 10).

As Algorithm 2 is written, we must expand the entire translation unit in memory. Unfortunately, the size of a typical translation unit in a real program can be large (200K characters is considered small nowadays). If expanding the entire translation unit in memory is not feasible, we can instead expand the translation unit into a file. That approach noticeably increases the time spent by Algorithm 2 performing file I/O. Alternatively, we can expand the translation unit in memory lazily, a piece at a time. The details of how to change Algorithm 2 to do lazy macro expansion are left to the reader.

Notice that Algorithm 2 returns the last node visited in the prefix tree. Given a node n in a prefix tree, we can reconstitute the tree of the corresponding translation unit by the following algorithm:

Algorithm 3

reconstitute(n) is

```
    R ← root of an empty parse tree
    while parent(n) ≠ nil
        for each level-one subtree t associated with n in reverse order
            make t the leftmost child of R
        n ← parent(n)
    return R
```

First we create a tree consisting only of a root node. Then beginning at the given node n we walk up the prefix tree. At each node we retrieve copies of the level-one subtrees associated with that node, and we splice them into the tree under construction. The only tricky part is ensuring that we splice the level-one subtrees in the correct order.

9 Complexity

In this section we analyze the asymptotic complexity of Algorithm 2. First consider running time. Let us make several reasonable assumptions. Let us assume that the time required to preprocess a translation unit is $O(L)$, where L is the number of characters in the macro-expanded translation unit. Let us also assume that *parseprefix* runs in time $O(len)$, where *len* is the first value in the triple (see Section 7) returned by *parseprefix*. Finally, let us assume that the time needed to apply Δ_m to P is proportional to the number of characters in TEXT_m.

Let us call a node m in the prefix tree *examined* if the value of TEXT_m is ever examined on line 5 of Algorithm 2. Let x be the number of nodes examined by the algorithm. Further, let s be the maximum number of characters in any top-level C++ statement contained in the translation unit being parsed or represented anywhere in the prefix tree. Finally, let l be the sum of all the values of

len returned by calls to *parseprefix*. Notice that we can implement the modifications to the variable e in Algorithm 2 with constant-time index manipulations; if we do so, then a straightforward analysis shows that the worst-case running time of Algorithm 2 is $O(L + sx + l) = O(L + sx)$.

In practice, x is bounded by a small constant. In well-written code, s is also bounded by a constant. Hence, in practice on well-written code the running time of Algorithm 2 is $O(L)$, or $O(l)$ not counting preprocessing time. This result is asymptotically optimal for any parsing algorithm that always preprocesses the entire translation unit.

Now consider space. Let us make the reasonable assumption that the memory needed to hold the *ptree* and Δ returned by *parseprefix* are $O(len)$, where *len* is the first value in the triple returned by *parseprefix*. If, immediately before lines 8 and 14 of Algorithm 2, we free the space needed to hold the node pointed to by n, then the space used by Algorithm 2 in the worst case is $O(L + s)$. If we also use file-based or lazy macro expansion as explained in the previous section, then the space used by Algorithm 2 is $O(s)$, which for well-written code is the asymptotically optimal $O(1)$.

10 Adding Transactions

A stored prefix tree is shared by many executions of *parse* and *reconstitute*. Only one execution at a time, however, should access the prefix tree. To prevent conflict, we can use the traditional technique of explicitly locking and unlocking the file containing the tree.

Alternatively, we might try storing the tree in a database, thereby getting serialization of accesses without having to do explicit locking. Specifically, the code for *parse* would look like this:

```
parse(t, T) is
        begin read-write transaction
        // ...
        commit transaction
```

Using a database has unfortunate implications for the efficiency of the algorithm, however. In many (perhaps all) databases, the memory used to hold a retrieved object cannot be freed until the current transaction is committed or aborted. Thus, in *parse* we cannot free the memory used to hold a node as soon as we leave that node.

One way to solve this problem is to commit and restart the transaction after examining every k nodes, for some constant k. There are several problems with this approach, however. First, it is difficult to implement (the details are omitted). Second, the resulting algorithm would be noticeably slower. And most important, the resulting algorithm would serialize incorrectly — that is, it would be possible for two overlapping executions e_1 and e_2 of *parse* to have an effect that is different from the effect of executing e_1 followed by e_2 or e_2 followed by e_1. (Demonstrating why is an exercise left to the reader.)

11 Current Implementation

To see how much of a speedup the algorithm described in this paper achieves in practice, we incorporated the algorithm (not using transactions) into an existing C++ parser. (This parser produces ALF trees [1].) To ensure that the experiments we conducted would be fair, we thoroughly profiled

and optimized the parser with the prefix analysis optimization both turned off and turned on. All tests were run on a relatively quiescent Sun IPX workstation.

We ran the parser on a number of files which, when macro-expanded, produced translation units of varying sizes. (The files were taken from the implementation of the parser itself.) Each file was first (1) parsed with the prefix analysis optimization turned off, then (2) parsed with the optimization turned on, then (3) parsed again with the optimization turned on. The results are shown in Figure 2.

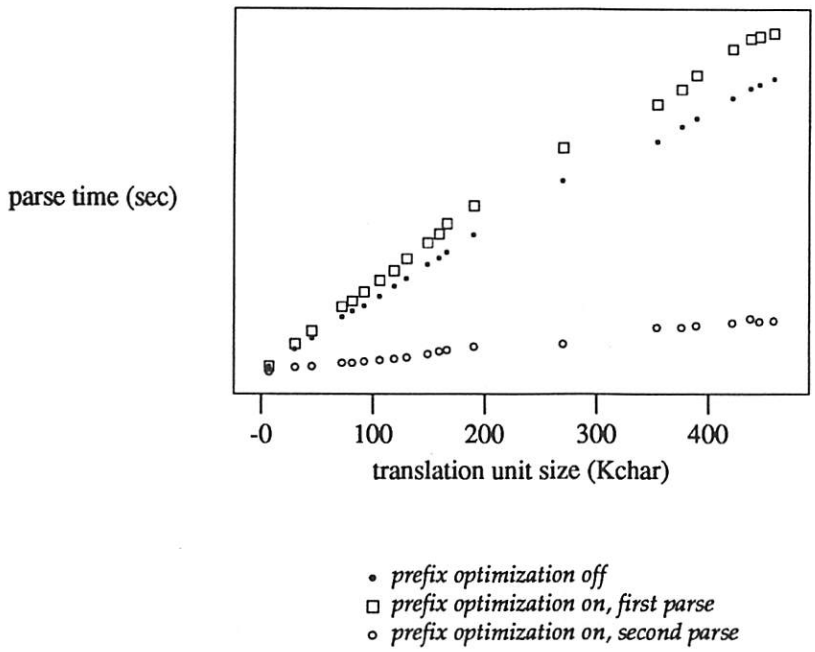


Figure 2: Parse times.

Notice that the first time that we parse with the prefix analysis optimization turned on, the parser is slower than with the optimization turned off. This behavior is to be expected: With the optimization turned on, the parser must spend extra time building and storing the prefix tree. The second time we parse with the optimization turned on, however, the parser is significantly faster than either of the other parses: The parser determines by consulting the prefix tree that none of the translation unit needs to be parsed again.

For large translation units, the first parse with the optimization turned on is 15–18% slower than parsing with the optimization turned off, and the second parse with the optimization turned on is 80–82% faster than parsing with the optimization turned off.

What is the effect on the parser speed if we make a change somewhere in the translation unit, then parse again with the optimization turned on? To answer this question, we chose one of the files f that produced a large (470 K) translation unit, made one noncommentary change to the code in a file that was directly or indirectly `#included` by f, and reparsed f. We performed this experiment several times. In Figure 3, we have plotted the resulting parser speed versus d, the distance (in thousands of characters) of the change from the beginning of the translation unit. Notice that the parser speed depends on d. This result is as we might expect: the farther the change from the beginning of the translation unit, the less code the parser has actually to parse. In particular, if the change is made in the dot-c file being parsed (as opposed to one of the included files), reparsing is quite fast.

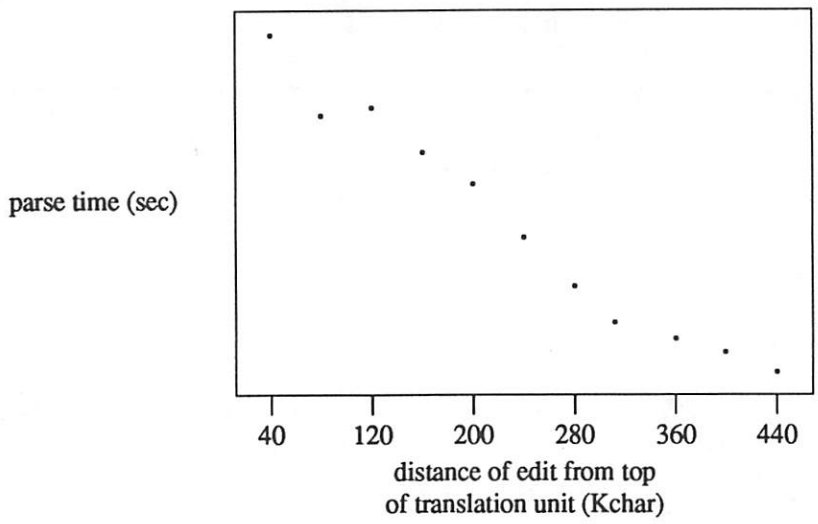


Figure 3: Parse times after edit.

12 Summary

In this paper, we showed how to use common prefix analysis to speed up a C++ parser. Our algorithm is efficient, near maximal, transparent, and relatively easy to implement. The key to the algorithm is to perform prefix analysis *after* preprocessing the code in the translation unit being parsed.

Acknowledgements This work grew out of the work of the Grail project at AT&T; thanks to my fellow project members — Per Abrahamsen, Steve Buroff, Peter Juhl, Andrew Koenig, Stan Lippman, Barbara Moo, Rob Murray, Bjarne Stroustrup, and Judy Ward — for their insightful conversations. Thanks in particular to Andrew Koenig for the original idea for the algorithm, and to Peter Juhl for implementing a major part of it. Thanks to Kiem-Phong Vo for his fast compression algorithm [6]. Thanks to an anonymous reviewer for his or her comment quoted in Section 3, and for urging me improve the section on precompiled headers.

References

[1] R Murray. 1992. A statically-typed abstract representation for C++ programs. In *Usenix C++ Conference Proceedings*, pages 83–98, August 1992.

[2] D. Franklin and B. Legget. 1993. Lucid Energize programming system for Sun SPARC. *The C++ Report*, 5(6), July–August.

[3] CenterLine Software, Inc. 1993. *ObjectCenter Reference.*

[4] M. Ellis and B. Stroustrup. 1990. *The Annotated C++ Reference Manual.* Addison-Wesley.

[5] G. Fowler. 1990. A case for `make`. *Software Practice and Experience*, 20:35–46, June.

[6] Kiem-Phong Vo. Unpublished compression algorithm.

Static Type Determination for C++*

Hemant D. Pande†
Tata Research Development and Design Centre
1 Mangaldas Road, Pune-411050, India

Barbara G. Ryder
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
ryder@cs.rutgers.edu

Abstract

Static type determination involves compile time calculation of the type of object a pointer may point to at a particular program point during some execution. We show that the problem of precise interprocedural type determination is NP-hard in the presence of inheritance, virtual methods and pointers. We highlight the significance of type determination in improving code efficiency and precision of other static analyses. We present a safe, approximate algorithm for C++ programs with single level pointers, using the conditional analysis technique [LR91]. We discuss the generalization of our approach to analyze programs with multiple levels of pointer dereferencing.

1 Introduction

Recent emphasis in the static analysis community has been on expanding compile time analysis to include interprocedural information [Bur90, Cal88, CBC93, MLR+93, CK88, CK89, HS90, HRB90, LRZ93, Mey81]. Historically, compile time analysis has been used in intraprocedural context for code optimizations. The emphasis is shifting towards including the use of interprocedural static analysis in all phases of the software life cycle including debugging, integration and testing [FW85, HS89, HRB90, Lak91, OW91, RW85, Wei84, YHR90]. However, until recently, software analysis tools either have not performed interprocedural static analysis or have employed grossly approximate techniques for languages with pointers. Landi and Ryder have shown the theoretical difficulty of static analysis in the presence of pointers and introduced a new technique for interprocedural analysis of C programs [LR91]. They have also developed a safe, approximate algorithm to solve the *aliasing* problem for a restricted subset of C which excludes pointers to functions, casting[1], union types, exception handling, *setjump* and *longjump* [LR92]. Arrays are treated as a single aggregate without distinguishing the individual elements. Our recent analysis of C programs [PRL91, PLR94], based on this work, represents one of the first attempts to obtain highly precise static interprocedural information for C programs and to apply it successfully in a software tool, the Test Analysis and Coverage Tool (TACTIC) [OW91].

Encouraged by the results obtained from analyzing C, we are now concentrating on how to employ the static analysis techniques beneficially to C++ programs. We have concentrated our efforts on developing new techniques to handle most of the features distinguishing C++ from C such as inheritance and virtual methods (object-orientedness), subtyping and overloading (polymorphism). The most significant C++ feature affecting compile time analysis is *virtual methods*. With virtual methods, it is the type of the receiver at an invocation site which dynamically determines the method to be invoked. With static *type determination*, such a late binding may be replaced by a function call to an appropriate method, or inlined code in suitable circumstances, thereby eliminating the overhead of late binding and improving the execution efficiency. Recent empirical studies of dynamic behavior

*This research was supported, in part, by funds from NSF grant CCR90-23628 and Siemens Corporate Research.

†Author's current address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903. email: pande@cs.rutgers.edu

[1] Although simple casting for `p = malloc()` is handled.

of actual C++ programs indicate there is opportunity to avoid late bindings in many cases [CG94]. Additionally, a statically determined list of possible types for a receiver would focus further analyses only on selected methods, rather than the entire pool of methods with the same name. Exclusion of the statically un-invocable methods from analysis would eliminate their spurious side effects, thereby improving the precision of subsequent analyses.

This paper describes initial results of our research in type determination. Ours is the *first* algorithm for type determination which uses the technique of data flow analysis without making gross approximations for the distinguishing C++ features mentioned above. The contributions of our work can be seen at two levels: (i) increased efficiency and precision of other *compile time* analyses and (ii) improved *run time* performance of the programs analyzed. In general, type determination cannot be done separately due to its interaction with aliasing. We present a type determination algorithm for the restricted case of single level pointers where the two problems are separable. Details of the generalized version appear in [PR94].

It is true that all C++ programs can be source-to-source transformed into C programs. Thus, if we claim to be able to analyze C, should we not be able to analyze C++ programs in their C incarnation? Actually, this is not desirable because the distinguishing C++ constructs map to C constructs so general that gross approximations in analysis would be inevitable. In particular, the virtual method mechanism can be expressed in terms of function calls through arrays of function pointers. Algorithms which attempt precise analysis in the presence function pointers and procedure variables handle only a limited usage of such constructs or resort to possibly worst case exponential analyses [Ghi92, HK92, Lak93, Ryd79]. This motivated us to develop new techniques to analyze virtual methods in the C++ domain itself; however, when there is no increase in generality, we reduce a C++ construct to a semantically equivalent C construct. For example, we transform a *class constructor* to a *malloc* followed by appropriate initializations and we express the principle of encapsulation using the concepts of scope and visibility in C.

Overview : In Section 2, we mention related work, especially for analysis of C++. We introduce the program representation, terminology and theoretical problem complexity in Section 3. We show that the problem is NP-hard in the presence of single level pointers; the intractability of the problem is inherent to this restricted case without generalizing to multiple level pointers. In Section 4, we describe a polynomial time algorithm to determine *points-to* information, i.e. the class of object pointed to by a pointer at a program point. We provide a running example to derive *points-to* values at some key program points and use it to bring out the significance of type determination. In Section 5 we briefly describe the interaction between type determination and aliasing in the general case. Finally, we conclude by summarizing our results.

2 Related Work

Program-point-specific type determination for object oriented languages has been attempted with varying degrees of success. Suzuki's algorithm [Suz81] handles languages like Smalltalk where objects serve as receivers of methods, but the problem is alleviated by the significant absence of pointers to objects. The algorithm by Palsberg and Schwartzbach [PS91] infers types of expressions in an object oriented language with inheritance, assignments and late bindings. They set up type constraints and compute the least solution in worst case exponential time. The algorithm does not perform control flow analysis nor does it track the values of objects. They suggest type determination using data flow analysis as an orthogonal way to aid their algorithm in performing optimizations and type safety checks. Recent work on improving run-time efficiency of the dynamically typed language SELF uses customization, iterative type analysis and inline caches to replace dynamic binding by procedure calls or inlined code [CU89, CU90, HCU91]. The algorithm by Larcheveque [Lar92] is rendered imprecise by the fact that it factors out the side effects of method invocations and aliasing due to parameter bindings as well as pointers. The suggested algorithms for these problems [CK89, Wei84] are grossly approximate and unsuitable in C++ context. We show that aliasing and type determination are inseparable in the general case, therefore a factored approach is not desirable. Ramesh Parameswaran has developed an algorithm which performs alias analysis without the knowledge of the receiver type

at an invocation site and thus assuming that all corresponding virtual methods are invocable [Par92]. He uses the precalculated alias information for type determination. Suedholt and Steigner [SS92] use a concept of *representant* virtual method to keep information about all the virtual methods with the same name. This approach leads to the loss of context which distinguishes one virtual method from others. Vitek *et al* [VHU92] present an algorithm which discovers the potential classes of objects for a simple object oriented language as well as a safe approximation to their lifetimes.

3 Problem Definition

Program Representation

A *control flow graph* (CFG) for a method consists of nodes which represent single-entry, single-exit regions of executable code and edges which represent possible execution branches between code regions. We represent a program with an interprocedural control flow graph (ICFG), which intuitively is the union of CFGs for the individual methods comprising the program [LR92]. Formally, an ICFG is a triple $(\mathcal{N}, \mathcal{E}, \rho)$ where $\mathcal{N}$ is the set of nodes, $\mathcal{E}$ is the set of edges and ρ is the *entry* node for `main`. $\mathcal{N}$ contains a node for each simple statement in the program, an *entry* and *exit* for each method, and a *call* and *return* node for each invocation site. An intraprocedural edge into a *call* node represents the execution flow into an invocation site, while an intraprocedural edge out of a *return* node represents control flow from an invocation site once the invoked method has returned. (We will use the terms *call* and *invocation* interchangeably.) For a non-virtual method call, we represent the control flow into the called method by an interprocedural edge from *call* to the corresponding *entry* node. Similarly, we represent the return of control from the called method by an interprocedural edge from the *exit* node to the *return* node. However, virtual method invocation makes it impossible to determine before analysis the correspondence between a *call* and *entry* since the method invoked depends on the type of the receiver at the call site. Establishing the interprocedural edge(s) from a *call* node representing virtual method invocation to appropriate *entry* node(s) and from *exit* node(s) to the *return* node is part of the algorithm presented in this paper.

Terminology

- We define an ICFG path from ρ as *realizable* if, whenever a method on this path returns, it returns to the call site which invoked it. Not all paths in the ICFG are realizable. Our analysis tries to restrict itself to realizable paths since unrealizable paths do not correspond to any possible execution sequence.
- *Objects* are locations that can store information, and *object names* provide ways to refer to them. An object name is a variable name and a possibly empty sequence of dereferences and member accesses.
- An *alias* occurs when there exists a realizable path to a program point, such that two or more names exist for the same location at that program point. We represent aliases by unordered pairs of object names (e.g. $<v, *p>$). The order is unimportant since aliases are symmetric.
- The *Type Determination Problem* involves calculating the type of the object pointed to by a pointer at a program point as a result of some execution that ends at that program point.
- A *pointer-type pair* $<p \Rightarrow C>$ *holds* on the realizable path $\rho n_1 n_2 \ldots n_i$ if p points to an object of class C after execution of program point n_i whenever the execution defined by that path occurs.

Theoretical Complexity of the Problem

Theorem 1 *In the presence of single level pointers and virtual functions in C++, precise program-point-specific type determination is NP-hard.*

We prove the theorem by a polynomial reduction of the NP-Complete problem of 3-Satisfiability to type determination in the presence of single level pointers and virtual functions in C++ [PR94]. □
An easy corollary follows, since the theorem involves a subproblem of the following problem.

Corollary *In the presence of multiple level pointers and virtual functions in C++, precise program-point-specific type determination is NP-hard.*

4 An Approximate Type Determination Algorithm

Our algorithm uses the idea of *conditional analysis* as found in [LR91]. Execution flow in a method is analyzed by assuming information that can hold at the entry of the method. Thus, in a sense, the resulting analysis is conditional on the assumed information at entry. The algorithm is rendered practical by doing computation only for those assumptions which actually reach the entry node on some execution path. The algorithm described here is applied only to C++ programs allowing a single level of dereferencing with pointers. We assume in the following description that the receiver of a method call is the first actual parameter and the corresponding formal is denoted by *this*.

We define a predicate *points-to* with the following interpretation: *points-to*$(n, assumption, fact)$ == *true* if (i) there exists a realizable path to the entry node of the procedure containing node n, on which assumption holds; and (ii) given that (i) is true, there exists a realizable path from the entry node to n on which *fact* holds. The assumption can either be $\emptyset$ or a pointer-type pair while *fact* is a pointer-type pair.

4.1 A Running Example

Before discussing the algorithm, we list a program segment in Figure 1. We will use it in Section 4.2 to illustrate the significance of type determination in practical issues of run-time efficiency and benefits to other optimizations. Throughout Section 4.3, we will use Figure 1 as a running example for the algorithm description.

4.2 Practical Issues

At node $n9$ in Figure 1, the pointer q is made to point to an object of class *Base*, and then immediately used at node $n10$ as the receiver for a virtual method invocation. Under these circumstances *Base* :: *foo*() will be invoked on all executions notwithstanding the virtual nature of the invocation. Since the virtuality of *Base* :: *foo*() is not utilized, the invocation can be compiled as a function call, thereby reducing the run time overhead of virtual invocation.

Limiting the scope of invocation to *Base* :: *foo*() and eliminating *Derived* :: *foo*() from consideration may benefit other analyses. The assignment at node $n2$ and printing *hello world* at $n3$ will not appear as possible side effects of the invocation at node $n10$. As another significant implication, our algorithm will be able to determine that $n11$ is a call of *Base* :: *bar*() and never *Derived* :: *bar*(), because the receiver a may only point to an object of type *Base*. Therefore, call site $n11$ can be considered non-virtual. Given the potential disparity in side effects of virtual methods which share the same name, type determination can significantly improve the precision of analysis.

Resolving a virtual method invocation to a unique function call may create possibilities for inlining, resulting in elimination of function call overhead. Inlining a function call can also provide opportunities for various intraprocedural optimizations.

A transformation from virtual invocation to function call is sometimes possible without complete resolution of the receiver type. For example at node $n12$, the receiver p may point to an object of class *Base* or *Derived*. Since the receiver type is not unique, a naive approach may result in retaining the invocation as virtual. However, since class *Derived* inherits the method *baz*() from class *Base* without redefining it, $n12$ may still be safely compiled as a function call to *Base* :: *baz*(). In general, even if the receiver at the virtual invocation site does not point to a unique class, but all the receiver types utilize the same virtual method, the virtual invocation may be compiled as a function call.

For architectures which use deep pipelining and speculative execution, the issue of accurate control flow prediction assumes significant importance. Using static type determination to replace virtual invocations with function calls, when the target method is known at compile time, would yield benefits comparable to those obtained by profile-based prediction for C++ [CG94].

4.3 Algorithm Description

To determine the type of an object a pointer variable may point to at a given program point, we perform a fixed point computation of the equations describing the C++ statement side effects on the

```
class Base {
public:
        virtual foo ( );
        virtual bar ( );
        virtual baz ( );
} *a, *b, *p, *q;

Base::foo ( ) {
   n1: a = new Base;
}

Base::bar ( ) {
    ...
}

Base::baz ( ) {
    ...
}
```

```
class Derived : public Base {
public:
        foo ( );
        bar ( );
} r, *s;

Derived::foo ( ) {
   n2: a = new Derived;
   n3: printf ("hello world\n");}
}

Derived::bar ( ) {
}
```

```
main ( ) {
    if (-)
        n4 : p = new Base;
    else
        n5 : p = new Derived;
    n6  : s = &r;
    if (-)
        n7 : s->Derived::bar ( );
    n8  : p->foo ( );
    n9  : q = new Base;
    n10 : q->foo ( );
    n11 : a->bar ( );
    n12 : p->baz ( );
}
```

Figure 1: Example of Type Determination Algorithm

predicate *points-to*, as described below. Underlying this analysis, we have a data flow framework defined on the simple *true/false* lattice. The elements of the lattice describe the values of *points-to* predicates at each program point. We present an algorithm which is both *safe* and *approximate*. If there exists a path to node n on which $<ptr \Rightarrow C>$ holds during some execution, our algorithm will report a *true* predicate *points-to*$(n, APT, <ptr \Rightarrow C>)$ for some APT, guaranteeing the safety of calculation. However, owing to the intractability of the problem, our polynomial time algorithm is justifiably approximate, reporting an overestimate of the actual solution.

We use a worklist for the fixed point computation. Whenever a predicate *points-to*(n, APT, PT) becomes *true* for the first time, it is placed on the worklist. Once marked *true*, a predicate stays *true*. Thus a *true* predicate goes on the worklist exactly once, guaranteeing the termination of our algorithm. We refer to this action as **make-true** and denote it in the algorithm by "**make-true** (*points-to*(n, APT, PT))".

We describe the algorithm in three phases: (i) we *initialize* the information, (ii) during the *introduction* phase we annotate each node appropriately with the information obtained locally at the node itself, and (iii) we *propagate* the information throughout the ICFG until stabilization. All *points-to* predicates are assumed *false* initially. For efficiency, we have designed the algorithm in such a way that the work is performed only for *points-to*$(n, APT, <ptr \Rightarrow C>)$ which are to become *true*. Given the solution for *points-to* at node n, the information about pointer-type pairs at n can be easily computed as follows:

for each node n in the ICFG
If n is

1. `n : p = new t :`
   make-true $(points\text{-}to(n, \emptyset, <p \Rightarrow t>))$
2. `n : p = &r :`
   make-true $(points\text{-}to(n, \emptyset, <p \Rightarrow type(r)>))$
   where $type(r)$ returns the type of object name r.
3. `n : foo(param`$_1$`, ..., param`$_k$`) :`
   make-true $(points\text{-}to(entry_{foo}, <p \Rightarrow type(r)>, <p \Rightarrow type(r)>))$
   where `param`$_i$ is of the form `&r` with pointer variable `p` as the corresponding formal, and the call is non-virtual.

Figure 2: Introduction Phase

while worklist is not EMPTY
    remove $(n, APT, <ptr \Rightarrow C>)$ from worklist
    if n is a call node
        type-implies-type-from-call $(call, APT, <ptr \Rightarrow C>)$
    else if n is an exit node
        type-implies-type-from-exit $(exit, APT, <ptr \Rightarrow C>)$
    else
        type-implies-type-through-other $(n, APT, <ptr \Rightarrow C>)$

Figure 3: Propagation Phase

$pointer\text{-}type\text{-}info(n) = \{ <ptr \Rightarrow C> \mid (\exists APT) points\text{-}to(n, APT, <ptr \Rightarrow C>) == true \}$.

Conceptually, we start with no information at any of the ICFG nodes by initializing each possible *points-to* predicate to *false*. We also initialize the worklist to EMPTY. The time complexity of the initialization of the entire *points-to* predicate may appear as proportional to the number of predicates possible, but we have a constant time initialization by following a lazy approach [LR92, PLR94].

The first entries in the worklist come from the introduction phase. During this phase we **make-true** certain predicates at a node by looking at the local information available in the node itself. Figure 2 lists the nodes examined in the introduction phase and their associated actions. Note that in item 3 we restrict ourselves to non-virtual method calls. Without the knowledge of the receiver type, we can make no educated guesses about the method invoked. We handle virtual method calls during the propagation phase.

Using the program segment in Figure 1, we list the following examples of type introduction. Since there exists a path $entry_{main}.n4$ at the end of which $<p \Rightarrow Base>$ holds without assuming any information at $entry_{main}$, using item 1,

make-true $points\text{-}to(n4, \emptyset, <p \Rightarrow Base>)$

Since there exists a path $entry_{main}.n5$ at the end of which $<p \Rightarrow Derived>$ holds without assuming any information at $entry_{main}$, using item 1 we also have

make-true $points\text{-}to(n5, \emptyset, <p \Rightarrow Derived>)$

At node $n6$, using item 2 and the fact that r is an object of class *Derived*,

make-true $points\text{-}to(n6, \emptyset, <s \Rightarrow Derived>)$

During the propagation phase, the worklist entries are processed one at a time. Processing a worklist entry implies propagating the effects of the pair PT holding at node n given the assumption

APT, to all the successors of the node n, and then removing the entry from the worklist. New entries which become *true* as a result of this action are placed on the worklist. The computation reaches a fixed point when the worklist becomes EMPTY. We describe this phase as a case analysis on the kind of logical successor of each worklist entry. Figure 3 illustrates the propagation phase at a high level with the help of three propagation functions: **type-implies-type-through-other**, **type-implies-type-from-call** and **type-implies-type-from-exit**. In the following discussion, we explicate the high level view by describing each propagation function.

type-implies-type-through-other(n, *APT*, $<ptr \Rightarrow C>$)
This function captures the intraprocedural aspects of type propagation as described in the following cases.

case 1: If successor is an assignment to *ptr*, `m : ptr = ...`, the given *points-to* does not propagate through m. Whatever *ptr* pointed to before node m was encountered is immaterial.

case 2: If successor is an assignment of *ptr* to a pointer variable other than *ptr*, with or without casting (within inheritance hierarchy): `m : ptr' = ptr;` or `m : ptr' = (Class E*) ptr;`:

make-true (*points-to*(m, *APT*, $<ptr' \Rightarrow C>$)) and **make-true** (*points-to*(m, *APT*, $<ptr \Rightarrow C>$)).

Type casting appears in the latter node so that the assignment is type-correct, but it is unimportant for our analysis since *ptr'* points to an object of class C irrespective of the cast type.

case 3: If successor node m neither defines nor uses the pointer variable *ptr*, then the type of *ptr* is preserved: **make-true** (*points-to*(m, *APT*, $<ptr \Rightarrow C>$)). This is a case of simple propagation of information without any change. In the example for type introduction, we inferred *true* values for *points-to*($n4, \emptyset, <p \Rightarrow Base>$) and *points-to*($n5, \emptyset, <p \Rightarrow Derived>$). Propagating this information to the successor node $n6$ which preserves the type of p, we **make-true** both

$$\textit{points-to}(n6, \emptyset, <p \Rightarrow Base>) \quad \text{and} \quad \textit{points-to}(n6, \emptyset, <p \Rightarrow Derived>)$$

Using further applications of **case 3**, the information at $n6$ propagates to its successors as

$$\begin{array}{ll} \textit{points-to}(call_{n7}, \emptyset, <p \Rightarrow Base>), & \textit{points-to}(call_{n8}, \emptyset, <p \Rightarrow Base>) \\ \textit{points-to}(call_{n7}, \emptyset, <p \Rightarrow Derived>), & \textit{points-to}(call_{n8}, \emptyset, <p \Rightarrow Derived>) \\ \textit{points-to}(call_{n7}, \emptyset, <s \Rightarrow Derived>), & \textit{points-to}(call_{n8}, \emptyset, <s \Rightarrow Derived>) \end{array}$$

type-implies-type-from-call (*call*, *APT*, $<ptr \Rightarrow C>$)
This function is responsible for propagating a pointer-type pair at the call site to appropriate *entry* and *return* nodes. We consider the following cases.

case 1: Propagation is simpler when the corresponding *entry* is readily known, typically when *call* represents a non-virtual method invocation. As we already saw, *points-to*($call_{n7}, \emptyset, <s \Rightarrow Derived>$). Since s is visible in the called method *Derived* :: *bar*(), we **make-true**

$$\textit{points-to}(entry_{Derived::bar}, <s \Rightarrow Derived>, <s \Rightarrow Derived>)$$

At the call site $n7$, s is the first actual parameter and corresponds to the formal *this* of *Derived* :: *bar*(). Since *points-to*($call_{n7}, \emptyset, <s \Rightarrow Derived>$) is *true*, we **make-true**

$$\textit{points-to}(entry_{Derived::bar}, <this \Rightarrow Derived>, <this \Rightarrow Derived>)$$

If *ptr* is not visible in the called method, the type pointed to by *ptr* cannot change[2]. In this case we propagate the predicate *points-to*($call$, *APT*, $<ptr \Rightarrow C>$) directly to the corresponding *return* node as *points-to*($return$, *APT*, $<ptr \Rightarrow C>$).

[2] This is true because we only have single level pointers.

case 2: Call is virtual. Suppose the call node is: `n : rec→fun ( )`.
The entry nodes to which the effects of the given worklist entry have to be propagated depend on the type(s) of objects the receiver *rec* may point to at the call site. Two circumstances are possible: (i) some typing information is already available at the virtual call site before resolving a method to be invocable (**case 2.1**), and (ii) a method is resolved to be invocable before all the typing information to be propagated has reached the virtual call site (**case 2.2**).

case 2.1: $ptr == rec$ (i.e. *ptr* is the same variable as the receiver *rec*)

1. The effect of this *points-to* needs to be propagated only to the method invocable when the receiver points to an object of class C. In the example, $\textit{points-to}(call_{n8}, \emptyset, <p \Rightarrow Base>)$ propagates to $entry_{Base::foo}$ as

   make-true $(\textit{points-to}(entry_{Base::foo}, <p \Rightarrow Base>, <p \Rightarrow Base>))$

   but not to $entry_{Derived::foo}$. On the other hand, $\textit{points-to}(call_{n8}, \emptyset, <p \Rightarrow Derived>)$ propagates to $entry_{Derived::foo}$ as

   make-true $(\textit{points-to}(entry_{Derived::foo}, <p \Rightarrow Derived>, <p \Rightarrow Derived>))$

   but not to $entry_{Base::foo}$.
2. The effects of other accumulated information at the call site are propagated through the appropriate method(s) as follows:
   a) If the method call involves passing of an object address $\&r$ as an actual to a pointer formal f: **make-true** $(\textit{points-to}(entry, <f \Rightarrow type(r)>, <f \Rightarrow type(r)>))$. Note that this case could not be handled in the introduction phase, as the invocability of this method from call node n was not known then.
   b) For each $\textit{points-to}(call, APT', <ptr' \Rightarrow E>)$ where $ptr' \neq rec$:
   We determine the corresponding entry and perform actions as in **case 1**. Thus while propagating $\textit{points-to}(call_{n10}, \emptyset, <q \Rightarrow Base>)$, we also propagate the predicate $\textit{points-to}(call_{n10}, \emptyset, <s \Rightarrow Derived>)$, already *true* at $call_{n10}$, with

   make-true $(\textit{points-to}(entry_{Base::foo}, <s \Rightarrow Derived>, <s \Rightarrow Derived>))$

case 2.2: *ptr* and *rec* are distinct variables:
Suppose the *points-to* information currently available about the receiver *rec* at the given call node is:

$$\textit{points-to}(call, APT1, <rec \Rightarrow C1>) = true \quad \text{and} \quad \textit{points-to}(call, APT2, <rec \Rightarrow C2>) = true$$

According to this information, the receiver *rec* may point to an object of type $C1$ or $C2$ at the call site depending on the execution path. So the virtual method call $rec \rightarrow fun()$ may lead to the invocation of two distinct virtual methods with name *fun*. Hence the effects of the given worklist entry need to be propagated to the entry nodes of each of these invocable methods. This is done in the same fashion as for **case 1**, considering one *entry* node at a time. In the example, suppose $\textit{points-to}(call_{n8}, \emptyset, <s \Rightarrow Derived>)$ is the candidate for propagation at $call_{n8}$. We have also seen that $\textit{points-to}(call_{n8}, \emptyset, <p \Rightarrow Base>)$ and $\textit{points-to}(call_{n8}, \emptyset, <p \Rightarrow Derived>)$ are *true* at $call_{n8}$ with receiver p. Thus there are two distinct methods $Base :: foo()$ and $Derived :: foo()$ which may be invoked at $call_{n8}$. As a result, we propagate $\textit{points-to}(call_{n8}, \emptyset, <s \Rightarrow Derived>)$ to the corresponding *entry* nodes using:

make-true $(\textit{points-to}(entry_{Base::foo}, <s \Rightarrow Derived>, <s \Rightarrow Derived>))$
make-true $(\textit{points-to}(entry_{Derived::foo}, <s \Rightarrow Derived>, <s \Rightarrow Derived>))$

If the pointer variable *ptr* is not visible in any one (or more) of these invocable methods, the predicate on the worklist propagates directly to the *return* node by **make-true** $(\textit{points-to}(return, APT, <ptr \Rightarrow C>))$.

type-implies-type-from-exit ($exit$, APT, $<ptr \Rightarrow C>$)
Lastly we describe how the type information propagates from *exit* node to the corresponding *return* node(s). Let *exit* be the exit node of a method $fun()$ and the return nodes corresponding to *exit* be $r_1, r_2, \cdots, r_k$ at the instant of processing this worklist entry. New return nodes may be added later, when the method is determined to be invocable from other virtual method call sites. We do not consider them at this time. As explained earlier, when a new virtual method is determined to be invocable from a call node we propagate the effects of this call from the exit of the called method to the return node corresponding to the call node [**case 2.1** of **type-implies-type-from-call**]. Let the call nodes corresponding to these return nodes be $c_1, c_2, \cdots, c_k$. We do the following for each return node r_i:

If *ptr* is not visible in the method containing the return node r_i, we take no propagation action. Since the variable itself goes out of scope, we do not need to know its type. However if *ptr* is visible in the method containing the return node r_i, we have the following cases:

case 1: If APT is non-$\emptyset$, implying that APT holds at *entry* in order that *ptr* points to an object of class C at *exit*. Each call node c_i responsible for imposing APT at *entry* in turn leads to $<ptr \Rightarrow C>$ holding at its corresponding return node r_i.
If APT is imposed at *entry* of the invoked method without requiring any *points-to* predicate at the call node c_i (i.e. *points-to*$(entry, APT, APT)$ was made *true* during introduction phase), we simply propagate $<ptr \Rightarrow C>$ to r_i. In this case: **make-true** (*points-to*$(r_i, \emptyset, <ptr \Rightarrow C>)$). On the other hand, suppose it took *points-to*(c_i, APT'', APT') to impose APT at the *entry*, we have: **make-true** (*points-to*$(r_i, APT'', <ptr \Rightarrow C>)$).

In our example, suppose we are propagating *points-to*$(exit_{Base::foo}, <p \Rightarrow Base>, <p \Rightarrow Base>)$. We have two return nodes *viz.* $return_{n8}$ and $return_{n10}$. Since it takes *points-to*$(call_{n8}, \emptyset, <p \Rightarrow Base>)$ to impose $<p \Rightarrow Base>$ at $entry_{Base::foo}$, using the information thus available at $call_{n8}$ and $exit_{Base::foo}$:

make-true (*points-to*$(return_{n8}, \emptyset, <p \Rightarrow Base>)$)

As there is no assignment to p on any path from $call_{n8}$ to $call_{n10}$, *points-to*$(call_{n10}, \emptyset, <p \Rightarrow Base>)$ is *true*. This predicate also imposes $<p \Rightarrow Base>$ at $entry_{Base::foo}$. Using this information available at $call_{n10}$ while propagating *points-to*$(exit_{Base::foo}, <p \Rightarrow Base>, <p \Rightarrow Base>)$:

make-true (*points-to*$(return_{n10}, \emptyset, <p \Rightarrow Base>)$)

case 2: If APT $== \emptyset$, implying that $<ptr \Rightarrow C>$ holds at *exit* without any assumption at *entry* of the method, we directly propagate $<ptr \Rightarrow C>$ to r_i using **make-true** (*points-to*$(r_i, \emptyset, <ptr \Rightarrow C>)$).

4.4 Algorithm Complexity

The following considerations are significant while determining the complexity of our algorithm.

1. The values of *points-to* are initialized in unit time (representation dependent).
2. The value of a predicate changes at most once, from *false* to *true*, and then stays *true*. A *true* predicate is only added to the worklist once, when its value has just been changed from *false* to *true*.
3. The total time complexity of actions performed for introductions and intraprocedural propagation is of the order of the number of ICFG edges, (or the number of ICFG nodes.)
4. For each ICFG node, the relevant solution is the third argument of the *points-to* predicate. For example, *points-to*$(n, APT1, <p \Rightarrow C>)$, *points-to*$(n, APT2, <p \Rightarrow C>)$ ...all yield the same inference that p may point to an object of class C at node n.

Assuming the above and further that the average number of assumptions (APT's) for each pointer-type pair derived at a node is bounded by a constant, the algorithm is linear in the solution size.

alias-implies-alias

$<*q, x>$
`n : p = q;`
$<*p, x>$

alias-implies-type

$<p, r>$
`n : p = &obj;`
$<*p \Rightarrow type(obj)>$

type-implies-type

$<q \Rightarrow C>$
`n : p = q;`
$<p \Rightarrow C>$

type-implies-alias

$<q \Rightarrow C>$
`n : p = q;`
$<p \rightarrow mem_i, q \rightarrow mem_i>$
$\forall$ member mem_i of class C

Figure 4: Intraprocedural type and alias propagation

5 Extension for multiple level pointers

In the presence of only single level pointers, a pointer cannot be aliased to another pointer. As a result, when a pointer changes its type (to point to an object of another type), it is only this pointer and nothing else which changes type. Type determination impinges on aliasing since the receiver types decide which virtual method is invoked at a call site, and the invoked method can affect aliasing. Aliasing plays no part in type determination. However such a separation does not occur in case of multiple level pointers. As an example, the node `m : p = &q` creates alias $<*p, q>$. Suppose subsequently on an execution path, `n : *p = &r` creates type pair $<*p \Rightarrow type(r)>$. In the absence of information that the alias pair $<*p, q>$ holds at node n, we would not be able to infer $points\text{-}to(n, \emptyset, <q \Rightarrow type(r)>)$, and the type determination would be rendered incorrect and unsafe. (Recall, it is unsafe to underestimate the set of possible types of a receiver object.) In Figure 4, we illustrate some intraprocedural aspects of the interaction between type determination and aliasing. The fact to be propagated appears first, followed by node n, and then we list an appropriate resulting fact.

Our algorithm described in Section 4.3 can extend to handle the general case of multiple level pointers, however it involves interleaved type determination and aliasing calculations. We have implemented a prototype for the general algorithm to perform type determination and aliasing together for C++ programs with multiple level pointer dereferencing. A detailed description of the general algorithm and preliminary implementation results can be found in [PR94]. Although current results are encouraging, we are extending the implementation to analyze a broader range of larger C++ programs in order to make a more definitive empirical assessment of our algorithm.

6 Conclusions

We have presented a polynomial-time approximate technique to perform program-point-specific, interprocedural type determination for C++. We have shown the theoretical difficulty of this problem and demonstrated the utility of its solution in virtual method name resolution. This is the first static analysis algorithm for type determination which accounts for pointers and virtual methods without gross approximations. For ease of explanation we have restricted the problem domain to C++ programs with only single level pointer dereferencing, where the virtual name resolution is separable from other analyses. We are currently gathering data from our implementation of the general algorithm to determine its practicality. We also plan to extend our work to solve other analysis problems useful for applications such as debugging and testing in a C++ programming environment.

Acknowledgements

This research benefited greatly from discussions with William Landi of Siemens Corporate Research. We also thank Rakesh Ghiya for his input in the design of this algorithm. We are indebted to Ashok Sreenivas, R. Venkatesh and Ulka Shrotri of TRDDC for their feedback on the algorithm and invaluable help in the implementation. Finally we thank Tata Consultancy Services, Pune for letting us use the MasterCraft C++ source code as front end for our prototype implementation.

References

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993.

[CG94] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the Twenty First Annual ACM Symposium on Principles of Programming Languages*, January 1994.

[CK88] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.

[CK89] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49-59, January 1989.

[CU89] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 146–160, June 1989.

[CU90] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47-56, June 1988.

[FW85] P. G. Frankl and E. J. Weyuker. A data flow testing tool. In *Proceedings of IEEE Softfair II*, December 1985.

[Ghi92] Rakesh Ghiya. Interprocedural analysis in the presence of function pointers. *McGill University School of Computer Science ACAPS Technical Memo 62*, December 1992.

[HCU91] U. Hölzle, C. Chambers and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object Oriented Programming*, July 1991.

[HK92] Mary Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 3, September 1992.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26-60. January 1990.

[HS89] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis and Verification Symposium*, pages 158-167, December 1989.

[HS90] M. J. Harrold and M. L. Soffa. Computation of interprocedural definition and use dependencies. In *Proceedings of the 1990 International Conference on Computer Languages*, pages 297-306, 1990.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 93-103, January 1991.

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235-248, June 1992.

[LRZ93] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.

[Lak91] Arun Lakhotia. Graph theoretic foundations of program slicing and integration. *The Center for Advanced Computer Studies, University of Southwestern Louisiana Technical Report CACS TR-91-5-5*, 1991.

[Lak93] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993.

[Lar92] J. M. Larcheveque. Interprocedural type propagation for object-oriented languages. In *proceedings of the Fourth European Symposium on Programming (ESPO'92)*, February 1992.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induce aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9), September 1993.

[Mey81] E. M. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, pages 219-230, January 1981.

[OW91] T. J. Ostrand and E. Weyuker. Data flow based test adequacy analysis for languages with pointers. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991.

[PLR94] H. D. Pande, W. Landi and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. To appear in *IEEE Transactions on Software Engineering*, April 1994.

[PRL91] H. D. Pande, B. G. Ryder and W. Landi. Interprocedural def-use associations in C programs. In *Proceedings of the 1991 Symposium on Software Testing, Analysis and Verification (TAV4)*, October 1991.

[PR94] H. D. Pande and B. G. Ryder. Static type determination and aliasing for C++ programs. *Technical Report, Laboratory of Computer Science Research, Rutgers University*, in preparation, 1994.

[PS91] Jens Palsberg and Michael Schwartzbach. Object-oriented type inference. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 146-161, October 1991.

[Par92] Ramesh Parameswaran. Interprocedural alias and type analysis for pointers. *Masters Thesis, Department of Computer Science, University of Wisconsin - Madison.* 1992.

[RW85] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367-375, April 1985.

[Ryd79] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–225, May 1979.

[SS92] Mario Suedholt and Christopher Steigner. On interprocedural data flow analysis for object oriented languages. In *Proceedings of the International Conference on Compiler Construction, Germany*, 1992.

[Suz81] Norihisa Suzuki. Inferring types in smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187-199, January 1981.

[VHU92] Jan Vitek, R. Nigel Harspool and James S. Uhl. Compile-time analysis of object oriented programs. In *Proceedings of the International Conference on Compiler Construction, Germany*, 1992.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984.

[YHR90] W. Yang, S. Horwitz and T. Reps. A program integration algorithm that accommodates semantic preserving transformations. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 133-143, December 1990.

Supporting Truly Object-Oriented Debugging of C++ Programs

James O. Coplien
AT&T Bell Laboratories
cope@research.att.com

Abstract

Most debuggers do not support an object-oriented debugging model. A debugger should be able to provide the view that each object is an independent entity with its own breakpoint behavior. We also would like the debugger to plant a breakpoint on the "right" member function when a polymorphic identifier is involved. The technology used in most C++ implementations does not support these features as well as the rich run-time environments commonly provided for symbolic languages. This paper introduces the need for such constructs, and presents algorithms that can be used to implement them in the framework of common symbolic debuggers.

1: Introduction

Though debuggers were among the first serious C++ programs written, [1] they are among the most anemic tools in most C++ environments today. Some vendors now offer good debuggers for specific hardware and operating system platforms, but applications building on niche platforms are often left at the mercy of their indigenous C debugger. Many developers use a hybrid C/C++ platform for C++ development, and are force to use a debugger with a C heritage to debug C++ code. In the pioneering days of C++, when most of the user base was tied to **cfront**-based compilers, installations that were serious about debugging taught their C debuggers some of the rudiments of C++. While the growing C++ market will increase the likelihood for more environments to enjoy full-fledged debuggers that support C++ language features, there will always be projects wishing to build C++ debugger support into their C debuggers for niche platforms.

Most C debuggers (or for that matter, debuggers for most procedural languages) support many of the same debugging constructs: planting breakpoints, examining variables, and dealing with scopes and activation records. The C culture shares a common model of procedural debugging, but the C++ world does not yet have a comparably mature "model of C++ debugging." This immaturity owes partly to the object-oriented nature of C++, but it also owes much to the implementation technology underlying most C++ implementations.

We can contrast the C model of debugging with the Smalltalk model. Smalltalk programs run in a rich environment full of information about the running code. A programmer can interrupt program execution at any time and look around at the state of the world. There is no debugger tool per se. Objects are self-describing and can be queried, exercised, and modified on-the-fly. Programmers interact with their programs at run time using the same constructs and abstractions they use during design and coding. In fact, the CRC card design technique is closely tied to the Smalltalk style of debugging by browsing an interrupted program, providing a "hypothetical programming" approach to design. [2] Exploring a program at run time is an important component of object-oriented design, and a good debugger can be an invaluable aid during such discovery.

C++, in the heritage of C, distances the symbol-rich compile-time environment from the run-time environment. For reasons of efficiency and compatibility with C, C++ objects are usually not self-describing. These differences in technology cause us to think of C++ debugging differently from the way we think of Smalltalk debugging. (In fact, the paper you are reading has been rejected at other conferences because the reviewers felt that C++ developers are doomed to debug at the assembly language level, so object-oriented debugging models for C++ are only of academic interest!)

Many features come to mind as we think about what it means to have a "model of C++ debugging." We can list these features in an informal order of increasing sophistication:

- *Demangling*, which lets the programmer talk to the debugger using source identifier names;
- *Overloading*, so the developer can distinguish between multiple functions of the same name;
- *Scope*, to recognize object and member function scope; for example, class member function breakpoints;
- *The Actors model*, where we associate member functions with the objects on whose behalf they execute;
- *Coarse-grain debugging* which would allow you to simultaneously set a breakpoint on all member functions of a given class or object;
- *Inheritance*, so base class member functions can be treated as though they are part of the derived class;
- *Intelligent formatting of user-defined types* when displaying their contents;
- *Debugging inline functions*, by providing breakpoint capabilities for them;
- *Object-oriented programming*, so the programmer can interchangeably treat objects of derived types as though they were instances of their common base type.

We find many of these features in even the most rudimentary C++ debuggers, but support for these features becomes less widespread as we descend the list. Many debuggers have only addressed class scoping and disambiguation of overloaded function names. Few C++ debuggers fundamentally change the model of debugging from that of C, and the C++ programmer is left to manipulate their C++ program in terms of the intermediate C code generated by **cfront**, or in terms of a C model of the object code from the C++ compiler. A great debugger would let programmers walk around in their running program with the same facility as they manipulate their C++ source; an ideal debugger would let programmers think about their programs in terms of object-oriented *design* abstractions. Smalltalk's environment lets the Smalltalk programmer approach that ideal; with work, we can make a C++ debugger smart enough to provide many of the same capabilities.

This paper discusses models for two aspects of object-oriented debugging: support for object autonomy, and support for genericity. The paper also details implementation strategies for each, with the hope that they will be useful to projects wishing to add C++ to their local C debugger. This paper draws from work on prototype and semi-production debuggers widely used for C++ development at AT&T.

2: Object Autonomy

To consider what is meant by debugging support for object autonomy, we can contrast run-time aspects of procedural and object-oriented languages. Programs in either kind of language have both a dynamic and a static structure. In procedural languages, the unit of program composition is the function. We seldom worry about a function's existence until it is called and an activation record for it appears on the stack. These activation records capture the dynamic structure of a program, and their content is the focus of much of our debugging effort. In languages without recursion (such as FORTRAN), run-time mappings between procedures and activation records are straightforward, since any procedure has at most one activation record open. The situation is slightly more complex in C, because it allows recursion and a given function may have an arbitrary number of activation records. However, use of recursion is more often the exception than the rule, and though most good debuggers have constructs for selecting from among a given function's activation records, they are seldom used.

The class is the major unit of program composition under the object paradigm. Member functions form the bulk of static C++ program structure, just as in C. Though member functions have activation records at run time, the run time focus is on objects rather than on local function data. Multiple objects of a given class might be extant, and it is important to the programmer to be able to distinguish between such objects and to be able to easily access any one of them. Compare this with how we view a procedural program, where multiple "instances" (activation records) of the primary programming abstraction (a procedure) are the usual case. This difference in views affects how a programmer debugs their program, and the debugger should support the prevailing view (Figure 1).

The difference between these views is one of emphasis and interpretation, not of implementation. At the level of design, the class view is better suited to modeling genericity; we will return to this in the next section. The object view, often called the Actors model, is better suited to modeling object autonomy.

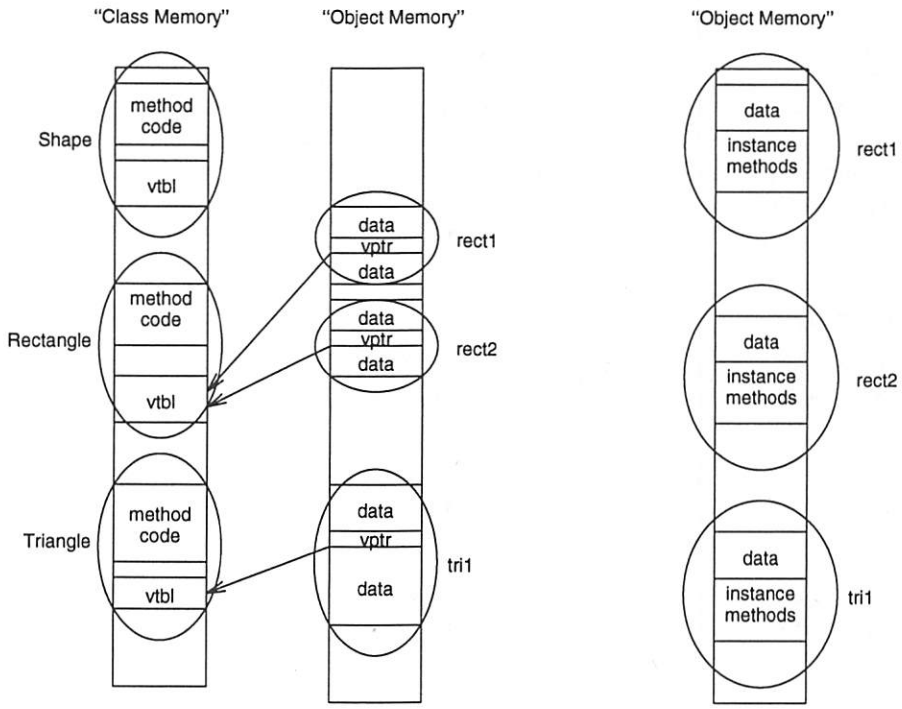


Figure 1. Object Implementation and User "Actor" View of Objects

A debugger command language might clearly distinguish between the class view and the object view. Consider the difference between:

```
when in Stack::pop { print "hello" }
```

and:

```
when in pancakes—>pop { print "hello" }
```

where `pancakes` points to an object of class `Stack`. The class form views the program in terms of its static structure, i.e., as a collection of classes, and the object form in terms of the objects making up its dynamic structure.

These two views are analogous to how one might view processes debugging in a multiuser programming environment. A simple operating system maintains a disjoint memory image for each invocation of a program, and each programmer has a complete copy of all text and data. If the environment supports shared text, special measures are necessary to allow one programmer to debug the code being executed by others. We would like to preserve both the illusion that each programmer owns their memory image, and the implementation advantages of shared text. The object-oriented analogue is for all objects to share the text of their classes' member functions, yet allow each object to be debugged as though it contained a complete copy of all its code.

We can easily teach a debugger to hide sharing details from the application programmer. The object breakpoint for `when in pancakes—>pop` can be formally defined as being equivalent to:

```
when in Stack::pop if (this == pancakes) { print("hello") }
```

where `this` is the anonymous object operand pointer argument passed to every member function. Such constructs were in fact used before the introduction of the object breakpoint construct in the debugger. The debugger's implementation of the object breakpoint construct can be obviously inferred from its class form.

The class view itself has two senses. The first sense is that of class methods (as opposed to instance methods), and is necessary in C++ to deal with member functions such as constructors. The other sense of

Class::Method understood by the debugger, particularly for instance methods, is that the associated operation (such as a breakpoint) applies to all instances of *Class* as though it had been applied to each individually.

Both the object and class views have been found to be useful in program debugging, and both approaches have been implemented in our prototype sdb++ debugger using a syntax culturally compatible with sdb. One breakpoint of each of these forms may simultaneously be active on the same line, with the object version taking precedence over the class version when both apply.

We implemented this technique in the framework of the existing C debugger, sdb, on a UNIX† SVR3 Operating System base using the AT&T/USL C++ Compilation System. The algorithms can be outlined as follows:

1. The debugger parses the command, recognizing it as a breakpoint command.
2. The operand is analyzed, and is discovered to indicate the name of an object pointer, and a function related to that object. The operand is parsed into those two components.
3. The symbol table is searched for the variable containing the object pointer. From that symbol table entry, perhaps with some additional searching using algorithms common to most symbolic debuggers, the *structure tag* of that variable can be found. In many implementations of C++, this structure tag names the original class of the original C++ text.
4. The name mapping scheme is applied to map the class name and function name onto a single name (e.g., `_move_9Rectangle_`), which is the mechanism used by the C++ compiler to fold the nested C++ name space onto the flatter C name space.
5. The symbol table is searched for the generated function name, and the function address is extracted from the symbol table entry.
6. The *breakpoint header table* is searched for an entry containing this function address. This is a simple table containing elements whose fields are: a function address, a saved op code for the location where an associated breakpoint or trap instruction is planted, and a breakpoint list pointer.

   The breakpoint list pointer points to a list of one or more *breakpoint structures*. A breakpoint structure entry contains the address of an object which must be the operand of this function invocation in order for this breakpoint to fire; a pointer to the next breakpoint structure, and a flag indicating whether the breakpoint is an Actors breakpoint or a ''regular'' (non-Actors) breakpoint.

   If a header table item is found whose address field matches the function address, skip to step 10; otherwise, proceed to step 7.
7. Create a new header table entry, and store the newly generated function address in the appropriate field.
8. Save the machine instruction at the generated address in the appropriate place in the newly generated table entry.
9. The machine instruction at the generated address is overwritten with an instruction that will cause a breakpoint to occur.
10. Search the breakpoint structure list for a matching entry; if one is found, this is a redundant breakpoint and gets exceptional handling (error or warning).
11. Create a new breakpoint structure. Designate this as an Actors breakpoint.
12. Taking the name of the variable containing the object pointer, go to the symbol table and find its address.

---

† UNIX is a Registered Trademark of UNIX† Systems Laboratories, Inc., a subsidiary of Novell, Inc., in the U.S. and other countries.

13. Go into the target process at this address, and retrieve the contents there. The result is the address of the object (operand) of interest.

14. Store that address in the appropriate field of the newly created breakpoint structure.

Now, when a breakpoint fires, the debugger searches the breakpoint header table for an entry with a matching address field. The debugger can also reach into the target process and retrieve the value of the address of the current operand—it is the value of the variable this from the current activation record. In the list associated with the identified breakpoint header table entry, the debugger searches for a match with that address. If such a match is found, the breakpoint is processed by keeping control in the debugger; i.e., not returning control to the application process except to temporarily restore the original op code and step the program over it, and then restore the trap instruction. If such a match is not found, then the original instruction is replaced and stepped, the breakpoint trap is replaced, and execution of the application process is resumed.

A limitation of this model is that breakpoints cannot be associated with variables before program execution, but can be associated only with extant objects after execution has begun. A breakpoint is not associated with an *identifier*, but with some *object*, though an identifier is used to indicate the object to which the breakpoint applies.

3: Support for Genericity

Genericity, and in particular, run-time type support, is a key concept in object-oriented programming. C++ supports this form of genericity through inheritance and virtual functions. This genericity makes it possible for a client of a group of objects to address any of them through a single identifier declared in terms of their base class. The client code sees only the base class interface and is thus insulated from changes to the form of derived class objects and even from addition of new derived classes.

We would like our debugger to give us the same derived class transparency. After all, the person using the debugger is likely the one who wrote the application code, and it is unreasonable to expect the programmer to know, at run time, the exact class of an object created from a hierarchy. To provide this transparency, the debugger must allow the programmer to communicate breakpoint and inspection requests in terms of a base class identifier, and resolve them itself in terms of the actual type of the object as determined at run time. That is, we want the debugger to be able to handle virtual functions with the same power and flexibility as understood by the C++ compilation system and the code it generates.

3.1: Type fields in disguise

How can the debugger determine an object's type? The naive answer would be to look in the symbol table. Unfortunately, most interesting objects in a C++ program are allocated from the heap, so they have no address that can be translated to an identifier, and hence to a type, using symbol table information. Another possibility would be for the debugger to retrieve the object's type from its memory image. Unfortunately, C++ objects have no type field information, at least none that is easy to find. In this respect C++ is a minimalist language, maintaining data structures as close to C as possible. The presence of a gratuitous type field would upset assumptions about the size and layout of class data. A compiler might annotate virtual function tables with type name strings, but most compilers do not: such strings would consume memory space, and would be useful only for classes having virtual functions. It is the lack of an explicit type field that makes this an interesting problem, and is why this is an issue in C++ and not in, say, Smalltalk.

In fact, the compiler does deposit a type field of sorts into each object whose class contains virtual functions—it must do so for the virtual function dispatching mechanism to work. Associated with each class is a table, commonly called the *vtbl*, most of whose content is function dispatching data. The compiler lays down an instance of this table for each class, and arranges for class constructors to deposit a pointer to this table in every object of that class. [3] This field might be viewed as a type field whose offset in the object can be determined from the class of the object.

In the polymorphic case, a derived class object is accessed through an identifier declared in terms of the base class (a reference or a pointer to the base class). The debugger is asked to manipulate the object (for example, plant a breakpoint) with respect to the identifier declared in terms of the base class. The base

class's *vtbl* pointer offset can be used even in the context of derived classes: The compiler guarantees that the same offset is preserved along the entire derivation chain.

3.2: Finding the function address

Now that the *vtbl* pointer offset is in hand, how do we find the address of the function? There is potential ambiguity if derived classes override the base class function having the same name as the one of interest. At this point, the debugger knows the name of the function that is to have a breakpoint associated with it, and it knows the address of the *vtbl* for the member function's class. If the debugger knew the class of the object of interest, and the function name, it could determine the breakpoint address directly from the symbol table. However, it does not know the class name at this point. Identifying the class's *vtbl* is not sufficient to identify the class, because a derived class may share its base class's *vtbl* if it overrides none of its base class virtual functions.

There is a roundabout way of determining the breakpoint address without ever having to know the name of the class, taking advantage of another property of *vtbl*s. Consider all classes in a given inheritance tree. Then for any function *foo*, the *vtbl* index at which *foo*'s entry appears is constant. All classes in an inheritance hierarchy maintain a constant mapping between function names (or, more precisely, function signatures) and their offset into the *vtbl*. We need to make one other assumption, which is that the requested function appears in the signature of the class corresponding to the identifier known to the debugger as the object's handle. This is a safe assumption: If it did not, the user would have no business addressing that function in terms of the given variable, and would be admonished with an error message.

We now gather information to support the algorithm that follows. The symbol table can be scanned for functions whose addresses match the addresses in the base class *vtbl* entries; i.e., those associated with the identifier supplied to the debugger by the user. When a match is found between the address of symbol table entry and a *vtbl* entry address, we can tabulate the *vtbl* entry's index with the symbol table entry's name. Most symbol table formats support the gathering of such information without the overhead of a linear search.

3.3: The complete algorithm for virtual function breakpoints

Now, we have our table of name/*vtbl* index mappings, the name of the desired function, and the locations of the desired object and, by consequence, of its *vtbl*. We can look up the *vtbl* index for our desired function from the table. Finally, we can use that index as an offset into the object's *vtbl*, and find the address of the function in that entry. At that address, we may plant a breakpoint.

Here are the details of the algorithm for the UNIX sdb-based debugger, given that context; they are also depicted in Figure 2.

1. The debugger parses the command, recognizing it as a breakpoint command.
2. The operand is analyzed, and is discovered to indicate the name of an object pointer, and a function related to that object. The operand is parsed into those two components.
3. The symbol table is searched for the variable containing the object pointer. From that symbol table entry, the structure tag (class name) is derived. This symbol table entry, generated by the compiler, contains compile-time known properties of the symbol, such as its type and its address in memory.‡ While the base type of the pointer is compile-time knowable, object-oriented languages allow such pointers to validly point to an object of any subclass of its declared base type, so something declared to point to a `Shape` may validly point to a `Rectangle` or a `Circle` at run time. So the class identified by this step is that corresponding to the pointer, *not* to the actual object it points to.

‡ For automatic (stack) variables, the symbol table usually contains an address field that corresponds to the symbol's offset within the activation record of the function in which it is declared; for external symbols, it is an address in the program's logical address space.

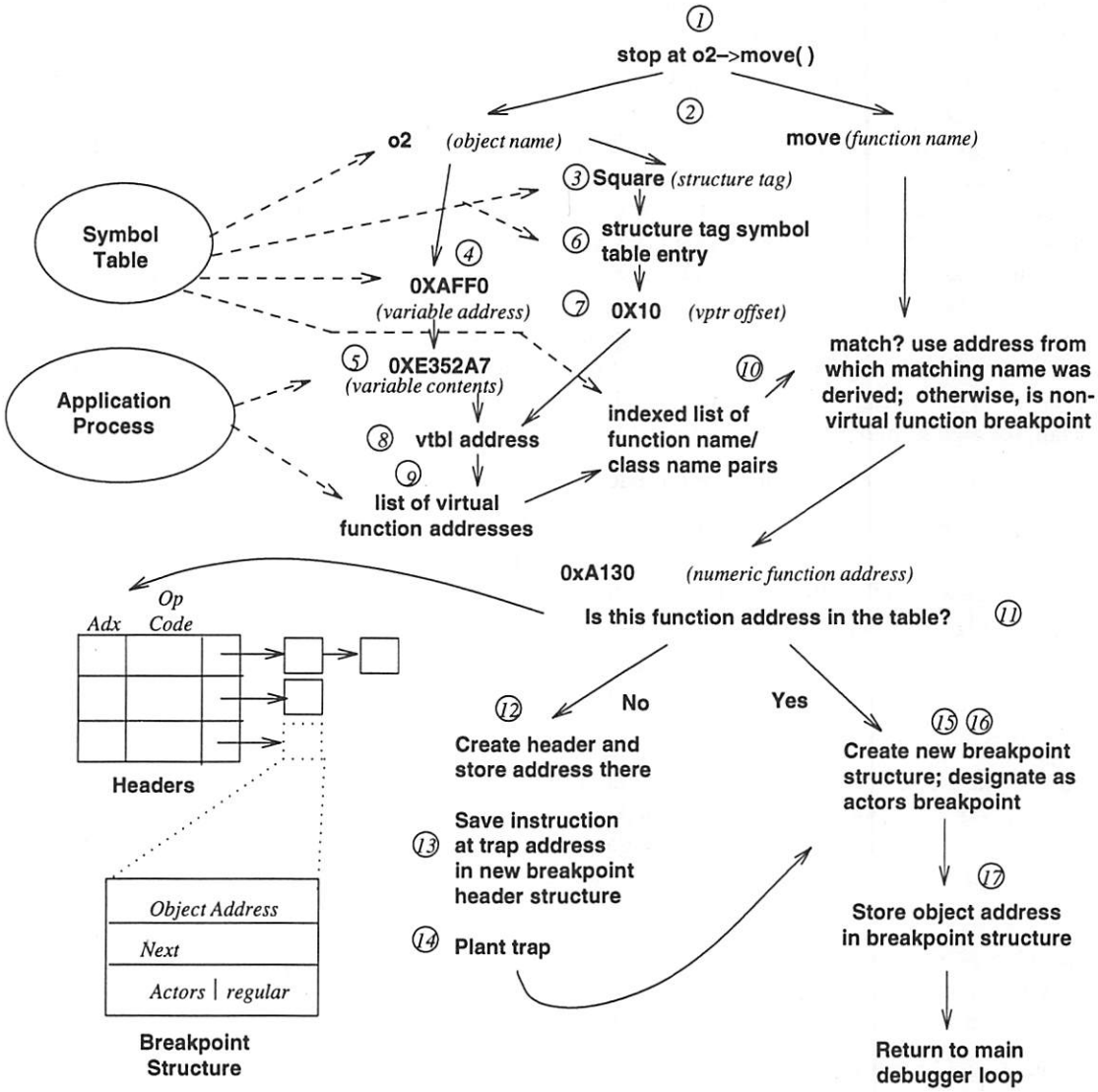


Figure 2. Virtual Actor Breakpoint Processing

4. From the same entry, extract the address of the operand variable.
5. Go into the target process at this address, and retrieve the contents there. The result is the address of the object (operand) of interest.
6. The symbol table is searched for an entry describing the fields of the structure named by the tag (class name) found above.
7. From that entry, the offset of the virtual function table pointer (`vptr`) is extracted.
8. Add this offset to the address of the object (operand) retrieved above, yielding the address of the word pointing to the `vtbl` associated with this object.

9. Retrieve the word at this address in the application program image; it is the address of the virtual function table (`vtbl`) for this object's class. The virtual function table is basically a list of pointers to functions; the index into such a table for a function of any given name is the same for all such tables and all such functions in classes participating in a derivation hierarchy whose root contains a function of that name. In reality, virtual function table entries contain additional data supporting multiple inheritance, which are ignored for the moment here.

10. The basic approach is to go through the virtual function table one element at a time, extracting function addresses from successive elements. Since function addresses are unique within a program, and function names are unique, there is a full, unambiguous and unqualified mapping back and forth between addresses and function names. However, in an object-oriented language like C++ that uses C technology for intermediate steps of the compilation process, the function names generated at the intermediate level may consist of two parts encoded together into a single name, those two parts being the class name and the function name.†† The encoding is reversible; the class and function names can be reconstituted unambiguously from the encoded name. We want to find the virtual function table entry whose function name component matches the function name specified by the user in step 2. This algorithm uses the object pointer supplied by the user, and information available from the object at run time, to deduce which function in the class hierarchy should be selected. In detail, the algorithm iterates over the virtual function table, and for each element does the following:

    a. Extract the function address for this entry.

    b. Do reverse symbolic resolution on the address; i.e., turn the address into a function name. This can be done by a linear scan of the symbol table (i.e., the task is tractable), but most debuggers build internal data structures to support doing this in a more efficient way; the details aren't important.

    c. Compare the function name component of the name/class pair thus generated with the name generated in the name parsing process from step 5 above. If there is a match, yield the address of the function and exit the loop.

    d. If the end of iteration is reached without finding a match, then this is *not* a virtual function Actors breakpoint, but a non-virtual function Actors breakpoint, and the ordinary Actors algorithm described earlier should be applied. Normally, the algorithm described here is applied first and, on this failure, the Actors algorithm is entered at the appropriate point (e.g., step 4) to avoid recalculation of data.

11. The address of the virtual function is now in hand. The breakpoint header table is searched for an entry containing this function address. If a header table item is found whose address field matches the function address, skip to step 15; otherwise, proceed to step 12.

12. Create a new header table entry, and store the newly generated function address in the appropriate field.

13. Save the machine instruction at the generated address in the appropriate place in the newly generated table entry.

14. The machine instruction at the generated address is overwritten with an instruction that will cause a breakpoint to occur.

15. Search the breakpoint structure list for a matching entry; if one is found, this is a redundant breakpoint and gets exceptional handling (error or warning).

---

†† The full signature (argument types, as well as the function name) is usually thus encoded into the internal (''mangled'') name by C++ translation systems.

16. Create a new breakpoint structure. Designate this as an Actors breakpoint.

17. Store the object address in the appropriate field of the newly created breakpoint structure.

4: Conclusion

The breakpoint techniques described here have been introduced in a number of prototype debuggers inside AT&T for a half dozen development platforms, and have been widely distributed throughout the company. The amount of effort needed to convert an existing debugger to use these techniques varies, but is a few staff-weeks on the average.

We have only anecdotal insights on how these techniques are used on AT&T projects. The Actors style breakpoints seem toenjoy frequent use, with generic breakpoints seeing somewhat less use. Part of this is due to the heavy use of data abstraction, but lesser use of inheritance, in projects actively using the debuggers. Debugger features supporting virtual functions and inheritance will likely see more use as understanding of the object paradigm deepens and spreads in the development community.

This algorithm was constructed in close collaboration with Tom Williams at Bell Laboratories. Discussions with Harold Bamford and Tim Born were also useful to refine our understanding of object-oriented debugging needs.

References

1. Cargill, T. A. "PI: A Case Study in Object-Oriented Programming." *SIGPLAN Notices 21(11)*, November 1986, pp. 350-60.

2. Personal communication with Kent Beck and Ward Cunningham, 1993.

3. Ellis, Margaret A., and B. Stroustrup. *The Annotated C++ Reference Manual.* Reading, Mass.: Addison-Wesley, ©1990, sect. 10.5.

HotWire -- A Visual Debugger for C++

Chris Laffra and Ashok Malhotra

I.B.M. Thomas J. Watson Research Center,
P.O.Box 704, Yorktown Heights, NY 10598, USA
Email: {laffra,petsa}@watson.ibm.com

ABSTRACT

We argue that visualization is essential in a modern debugger. Instead of inserting debug statements throughout the code, it should be possible to easily define visualizations while running the program under control of the debugger, resulting in what might be called "visual printf's". A visualization of a C++ program can provide exciting insights. Bugs that cannot be found that easily with non-visual techniques are now found, just by watching the visualizations. However, the mechanisms to define the visualizations should be easy to understand, easy to apply and cause only minimal overhead to the programmer (who is the *end-user* of the visual debugger). HotWire is not only equipped with a couple of standard visualizations, but also with a small declarative script language (using constraints) that can be used to define new custom visualizations. This paper addresses user interface aspects of debugging tools. Specifically, the user interface of **HotWire**, a debugger for C++ and SmallTalk on AIX and OS/2 is described.

INTRODUCTION

Every programmer knows that debugging is very time consuming, yet it is universally considered to be a second-class activity. Programmers hate to admit that they create bugs. "Real" programmers don't use a debugger. As a result, the development of debuggers is regarded as a third-class activity. Even though object-oriented (OO) techniques have been around for quite a while, debuggers for OO languages, and in particular C++, are still very immature. Most of them originate from debuggers for C. As a result, they use wrong metaphors, like structures and procedures. They do not recognize objects as objects. This is the typical C++ debugging scenario: Our C++ program crashes. We use a C++ debugger to determine where the illegal instruction occurred. After the location has been determined (the call stack is inspected), we may inspect one or two variables. But, the call stack is what we are after. We leave the debugger, edit the source code, insert some print statements, recompile the program, and run it. We repeat this (long) cycle, until we find the bug. Then we carefully have to remove the scaffolding to cleanup the source code again. To save time, C++ code is very often scrambled with #IFDEF DEBUG statements to use the debugging statements at a later stage. It is sad, but this is the case is most C++ program development today. However, we even know of C++ systems that take 5 hours for a complete compilation, and that cannot be loaded into a debugger within reasonable time. How do these people debug their C++ systems?

What is required, are filtering techniques that enable us to define and regulate our *focus* while we are trying to understand why a system behaves as it apparently does. Additionally, the enormous amount of information encountered while inspecting running applications should be represented in a fashion that allows programmers to actually interpret and use it. Breakpoint analysis, code stepping, or execution traces may indicate and solve bugs, but they typically do not work in real-world scenario's. It is senseless to try and inspect a megabytes sized trace file.

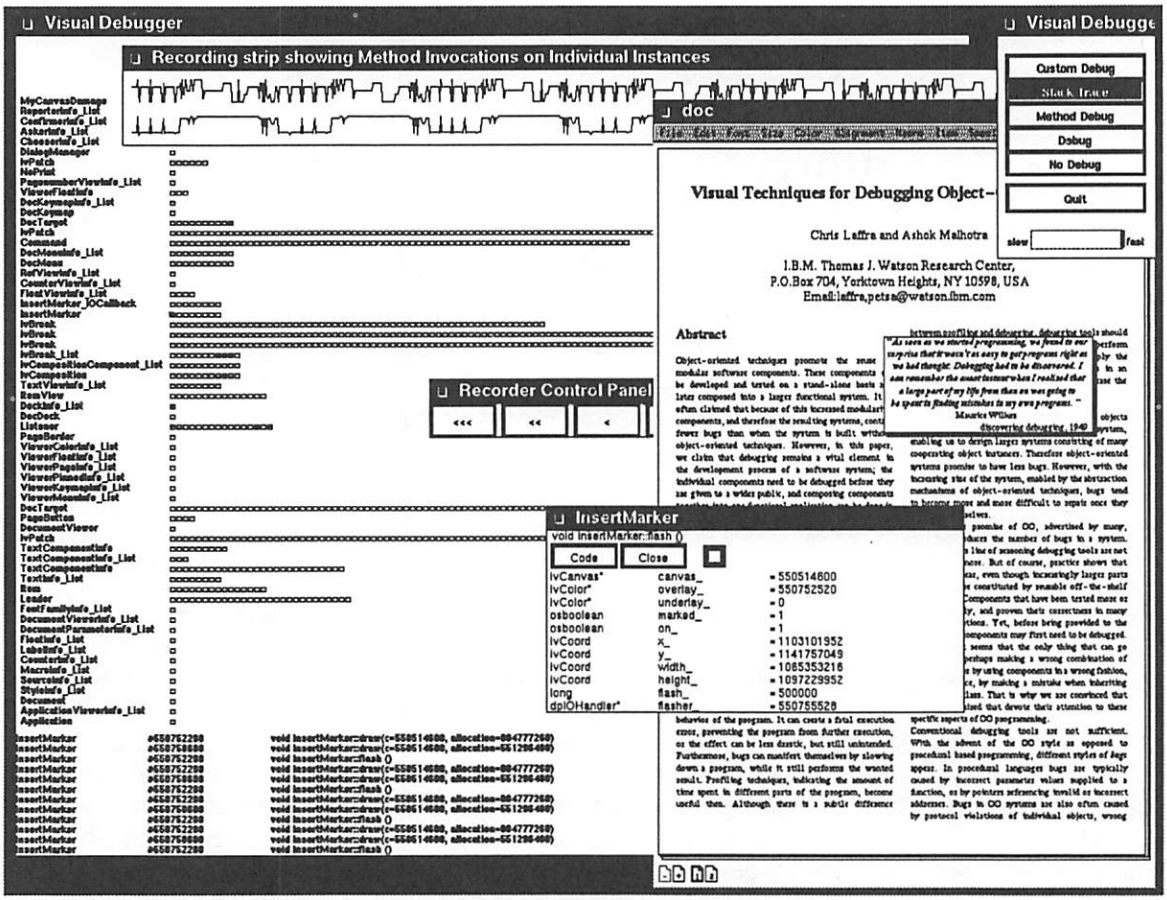


Figure 1: *A sample session with HotWire. The application shown in the window with the label "doc" (the large window on the right) is a wordprocessor. The other windows are part of the debugger. The large window on the left shows all instances in the application collected by type name. Each rectangle represents one instance. The rectangles are animated when a method is executed.*

We need abstraction techniques, showing the same information in another, more informative, way. This is where *visualization* comes into the picture. Visualization is the use of computerized images to enhance the insight of human beings into complex phenomena. Visualizations provide us with a way to improve the signal/noise ratio. Maps are one well-known abstraction mechanism with a very high information content [Miya 93]. Their signal/noise ratio is better than that of photographs or movies. The consistent characteristics of maps, such as easily measurable geometry, the use of symbols (always open to interpretation, though), and legends enable quick determination of source and destination. Aerial photographs are certainly less suitable for this purpose. The challenge is to find similar abstractions for displaying running OO programs. What techniques can be used to draw the map? What legends are required? What symbols are suitable? How can we change the resolution of the map and update the different abstractions and symbols on the map while zooming in (first look at the US, then at the Northeast, then at New York City, and finally at the streetmap of Manhattan). This hierarchical approach is one way to solve the problem of managing the information. But, there are many more.

Visualization techniques can help programmers by allowing them to detect anomalies in object activation and message patterns, simply by recognizing a "weird" aspect of a visualization. Visualizations offer a better representation of spatial relationships between objects, allows users to understand the picture "at-a-glance", can show trends and anomalies, and use cognitive associations (the viewer's pattern recognition skills). These points were recognized in earlier work, such as Balsa [Brown 88], Tango [Stasko 90], and the Object Visualizer [De Pauw 93]. Sound helps too [Smith 87], in that respect that incorrect behavior in the program

can be detected by recognizing that the program makes a "strange sound" (an important technique also used by car mechanics). Once the problem has been detected, it needs to be identified. Browsing techniques and more specific visualizations need to be applied to identify which particular part of the application is leading to the unexpected behavior.

Different applications and different situations require different approaches and visualizations to understand what goes on within the system, and how certain events are related to others. Debuggers should be equipped with built-in visualization techniques, but, more importantly, with facilities that allow for on-the-fly definition of visualizations, depending on the particular needs of the development team at that specific time. Furthermore, the mechanisms to define the visualizations should be easy to understand, easy to apply, and with minimal overhead.

As an aside, C++ compilers can make it more complicated by not generating enough information to allow C++ debuggers to do a decent job. People will have to become aware of the fact that debugging is an important aspect of program development, and that debuggers have a right to exist. A dialogue should be started between C++ compiler teams, debugger builders, and human factors experts.

In this paper we describe a visual debugger called **HotWire**. The debugger is equipped with fixed visualizations showing a selection of all object instances in the application and the method calls that are invoked on them (see Figure 1). In addition, the debugger is equipped with a special script language to define customized visualizations.

Before introducing the design issues that led to HotWire, we discuss specific type of bugs that could be introduced while using OO techniques. Then, some interesting visualization techniques in the context of OO debugging will be highlighted. It will be shown how they could enhance the understanding of the behavior of a program being debugged. Furthermore, different types of mappings that can be envisioned between the OO concepts and the visualizations will be discussed. Using visual techniques in the interface of a debugger enhances the understanding of a program. Understanding the program is the first step in detecting and identifying a malfunction.

WHAT CAN GO WRONG IN OO PROGRAMS?

Many problems in applying OO are related to the *syntax* of the language. Using the wrong type of brackets, omitting separators, etc. Most of these problems will be handled by the compiler, but some remain undetected until runtime. This is the fact with the list of classic bugs for Smalltalk [Johnson 93], which lists 12 problems with using Smalltalk. From that list, 6 bugs are directly related to the Smalltalk-80 environment. The rest are related to syntax, scope of local variables, object initialization, and different Smalltalk engines. In fact, only one addresses a problem that can be generalized to OO in general, and is not related to syntax or programming environment. It has to do with handling the Collection class.

When a given OO language is an extension of an existing language (like C), all problems that can occur in that underlying language, are automatically inherited in the OO language. Division by zero, loops that are traversed incorrectly, array bound violations, and invalid pointer arithmetic. These are some of the problems that debuggers may still need to take care of.

If we concentrate on OO concepts, the following things may go wrong, amongst others:

- Creating too many instances to do the job. There are a number of different forms of this problem:
  - Creating too many static objects. Normally, each block of code may contain the declaration of a local variable. With each execution of the block (it may be contained in a loop construct), the object is created, initialized, cleaned up, and removed. Typically, inefficiency is the result. Debuggers showing all instances dynamically, at runtime, address this problem (see Figure 2). Namely, the representations for the instances are created and deleted in a rapid rate. See also [De Pauw 93] for other interesting ways in which object histograms can be used.

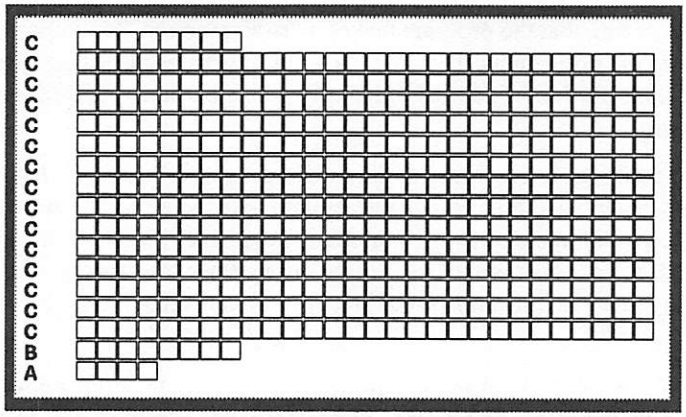


Figure 2: *This HotWire view shows all the instances in a given program. For each instance of the type* **A** *there exist two instances of the type* **B**. *There exist a large number of instances of the type* **C**, *which might indicate a memory leak.*

- Object groups. In many design patterns, objects are created in cooperating roles. An example could be that for each instance of the type A there should be two instances of the type B. Running the application, stopping the application at a particular moment in time, and counting the object instances shown in the debugger may reveal unexpected objects (see Figure 2).
- Unused objects instances (which cannot be collected as "garbage"). An application can create objects, and never actually use them at all during the execution. During development, some particular use was envisioned, and later changes in the program made these objects inappropriate. Debuggers that also show the activation on these objects (see Figure 3), might reveal this particular aspect, by highlighting at the end of the run all unused objects, and show where they were created. The debugger should remember where in the code instances are being created, so that selection of their representation will report that location.

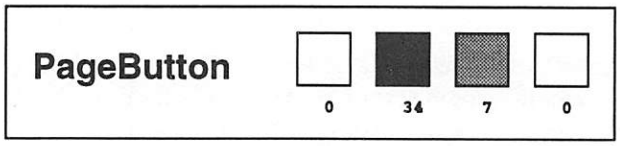


Figure 3: *This view shows that a given program contains 4 instances of the type* **PageButton**. *Instances 1 and 4 have never been used after creation, while instances 2 and 3 have until now executed 34 and 7 methods respectively. Instances 1 and 4 may very well be redundant objects. The shading of the instance also reflects the number of methods that have been invoked on the instance (shading poses a problem, in that there are only a limited amount of distinguishable shades).*

- Forgetting to delete instances. Objects may be explicitly created, manipulated, and passed around. After they serve their purpose, they should be deleted again. If they are not deleted, in languages without garbage collection, this results in memory leaks. A visual debugger indicates this problem. If you run your application long enough, it will simply fill up the debugger windows.
- Calling methods on the wrong target instance. For example, an instance can have references to a number of objects that perform some processing for it. Even though it may have been the intention of the programmer to equally divide the work among the different workers, a bug may result in having one single object doing all the work. Looking at the visualization of the method invocations shown in a visual debugger indicates this behavior (see Figure 4).
- Sending incorrect message contents to objects. Even when the target of a message is the correct object instance, and the actual method is the one that should be called, the contents of the method can still

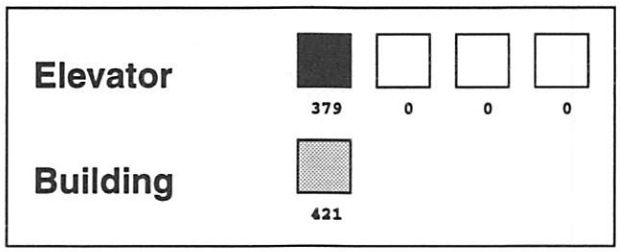


Figure 4: *This view shows a building with 4 elevators. It is clear that elevator 1 has done all the work, while elevator 2, 3, and 4 have done nothing so far.*

be invalid. Typical problems are passing references to wrong objects, wrong numerical values, null pointers, etc. Some problems directly lead to a program crash (rather simple to detect), but some may result in an incorrect program behavior (a lot harder to detect). A good OO debugger should remember the history of method calls, and should allow the programmer to browse through this recording to find incorrect calls. Figure 5 shows a snapshot where a call is made to a method with unexpected arguments.

- Sending messages to objects in the wrong sequence. Obeying the *protocol* of an object class or the combined protocol of a set of classes can be very difficult. In most OO languages, the programmer has no provisions to clearly specify the protocol of a class. When languages provide this facility (like the Protocol in Procol[van den Bos and Laffra 89]), it would ease the task of the programmer by providing a concrete specification of the access behavior of the object in question. Furthermore, if the specification is accessible at runtime, debuggers would be able to handle this very common access control problem much better, especially when the protocol could be visualized. As an alternative, keeping histories of method accesses may trigger the programmer to the occurrence of a protocol violation (see Figure 6). An important aspect of this problem is that it is especially apparent when using library classes. These toolkits typically consist of hundreds of classes, with thousands of methods. Using toolkits correctly is a major exercise and debuggers still have big difficulties in supporting this style of program development.

- Incorrect updates to the state inside a method. Often, the intended side-effect of a method is to update the state of the instance. Of course, this can be, and often is, done erroneously. Assertions over the state of the object, and pre-, and post-conditions (such as provided in Eiffel[Meyer 88]), can address this problem. But, simple visualizations of the state of the object are equally powerful, more versatile and dynamic, and more effective (see also Figure 10).

MAPPING OO CONCEPTS TO VISUALIZATIONS

The interesting question to be discussed next is how to connect the two very different universes of (1) OO concepts, such as classes, instances, and methods, and (2) custom visualizations, such as boxes, lines, texts, diagrams, and bitmaps, The form in which the mapping between these two worlds manifests itself is extremely important, as it will define the success of any visual debugger.

One way to define the mapping is to use the Model View Controller (MVC) paradigm [Goldberg 83]. The MVC concept is used in the Smalltalk environment to define user interfaces. The MVC triad separates the three components – functionality, input, and output. The underlying functionality is represented by a so-called *model*. The programmer can attach a *controller* and multiple *views* to each model. The view only contains the methods to make the model visible and to interact with it. The controller is used to control the communication between the views and the model. It ensures that each of the views always correctly represents the state of the model. In the context of this paper the model would reside in the program, and the views would be provided by the visualization library. The difficult part is defining the controller. If we are able to define a simple way to define controllers, it is relatively affordable to connect new views to models in our target program.

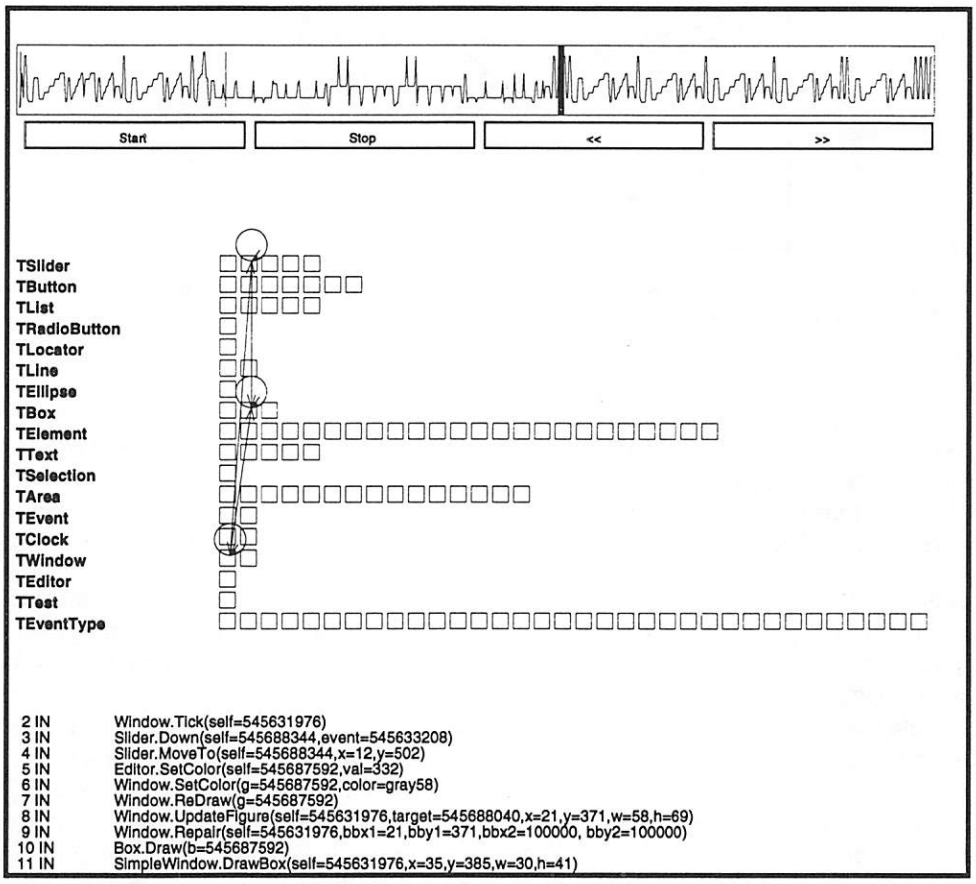


Figure 5: *This snapshot shows a debugging session of a drawing editor and debugger. It indicates a call to the method* **Window.Repair** *on line 9, with unexpected values for arguments* **bbx2**, *and* **bby2**. *The recording strip shows a history of methodcalls. Each pixel from left to right is one call. The height of the pattern is defined by the address of the instance. Each instance always appears at the same height. At the bottom, the callstack is shown at the selected time frame. The window in the middle shows all instances that were involved in the interaction pattern at that time.*

While the mapping between model and view in the MVC paradigm is defined in a *procedural* fashion, ideally the mapping should be declarative. A declarative mapping is easier to define, simpler to read, and understand. Furthermore, if the mapping is to be done in a visual fashion, a declarative "language" is also to be preferred. One could make the connections by simply clicking on objects or classes in the application and select an appropriate view from the library, again with a few mouse clicks. Once the mapping is given, the visualization system should maintain the relationship thus defined. Keeping the views up-to-date with their model is now a task for the visualization system, *not* for the programmer.

The development of a visualization or animation always requires a certain investment. Sometimes, the development of a specialized visualization is not justified, because other techniques allow for faster determination of a problem. However, when projects get larger, the initial effort of developing a visualization may prove to be very worthwhile at a later stage, either to test further developments of the system (effectively re-running the script from time to time), to report progress to third parties, or to educate new-comers to the project team.

Figure 7 shows a custom visualization of an elevator simulator. The emphasis here is on showing how simple it is to develop a visualization without the need of adding any extra statements to the code of the program. The visualization is entirely independent of the program.

Defining a visualization involves a number of steps. Step 1 is the identification of the elements of interest in the application (the models). Step 2 is the selection of appropriate visualizations. Step 3 is the mapping

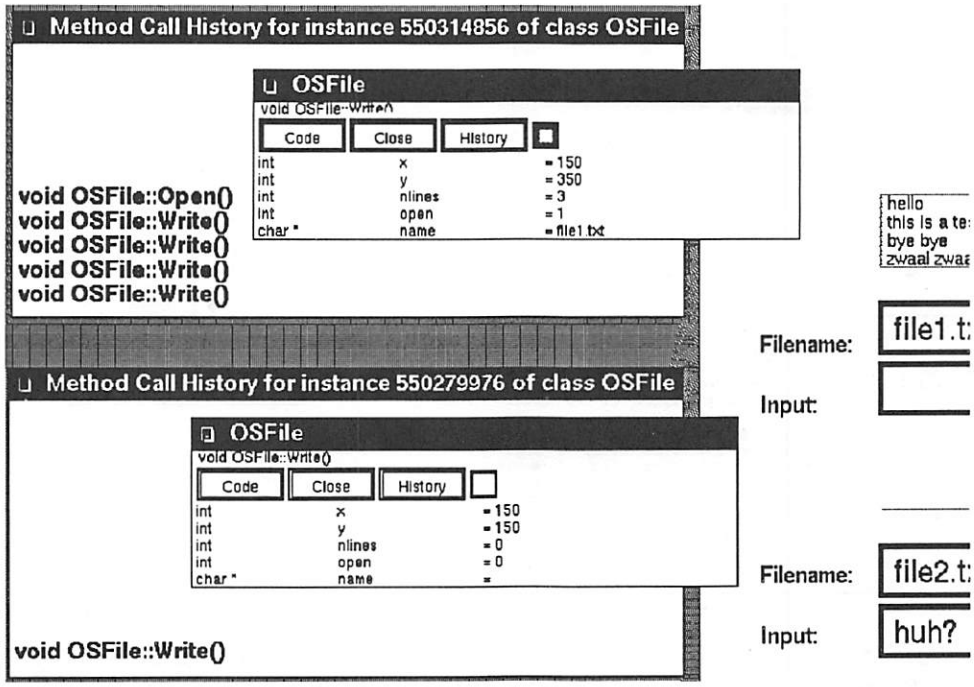


Figure 6: *This HotWire debugging session shows how keeping a history of method invocations can help identify object protocol violations. Two instances of the type* **OSFile** *are shown. Both are abstractions of the file concept. By typing in a name in the* **Filename:** *field, a particular file can be opened for writing. Then for each string that is given in the* **Input** *field, a line is written to the file. In the bottom interactor, the user forgot to hit the enter key, after typing in the filename to be opened. The instance browser in the left bottom reflects this fact, as no call to* **Open** *is recorded. Furthermore, the value of the* **open** *field is equal to 0, and not 1, as is the case in the top left window.*

between the models and their views. Step 4, the most difficult part yet, is to define spatial coordinates for the views, both absolute to the addressing space of the visualization environment, but also relative to each other. In most visualizations the relative layout of views is important, and their computation typically involves tedious calculations. The programmer has to come up with a translation from abstract relationships, such as *A must be left from B*, *C has to be inside D*, etc., into explicit coordinates to move individual views. Combinations of different relationships complicates matters even further. A different specification technique is required.

A powerful mechanism to define relationships is given by the *constraint* programming metaphor [Leler 88, Freeman-Benson 90]. Relationships are defined once, and the constraint solver will maintain the universe of constraints, ensure that all constraints are satisfied, and take measures when a certain condition invalidates a given constraint. Constraints can be one-way (working in one direction only), or two-way (updating both components, when one is changed) , see [Sanella 93]. In the current version of HotWire, one-way constraints are used to define relationship between graphical elements. Changes in the state are forward to dependents, which may again trigger other constraints. Circular dependencies are not dealt with perfectly. We are planning to incorporate a multi-way constraint system to handle the constraints in our specification language.

The following script shows how constraints are used to layout the bars in a bargraph animation for a sorting algorithm on a linked list. The linked list class is called `List` and has instance variables `value` and `next`:

```
class List {
public:
    int value;
    List *next;
    void SetValue(int value);
```

```
class Request {
public:
int floor;
...
};

class Elevator {
public:
int current_floor, target_floor;
int status;
...
};

class Building {
public:
Elevator *lifts[NLIFTS];
...
};
```

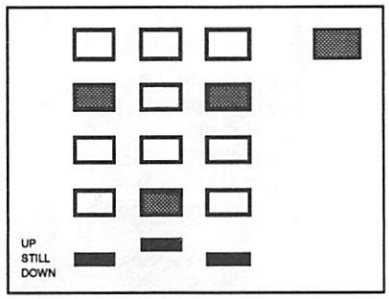
(a)

```
GLOBAL   window     Window 400 400 600 500
GLOBAL   x          Integer 100
GLOBAL   up         Text window 20 60 "Helvetica" 12 "black" "UP"
GLOBAL   still      Text window 20 40 "Helvetica" 12 "black" "STILL"
GLOBAL   down       Text window 20 20 "Helvetica" 12 "black" "DOWN"

TYPE     Elevator   Box window @x 100 60 50 "black" 2 "None"
TYPE     Elevator   Box window @x 200 60 50 "black" 2 "None"
TYPE     Elevator   Box window @x 300 60 50 "black" 2 "None"
TYPE     Elevator   Box window @x 400 60 50 "black" 2 "None"
TYPE     Elevator   Box window @x 60-status*20 60 12 "blue" 1 "blue"
TYPE     Elevator   Box window @x+3 3+current_floor*100 54 44 "red" 1 "red"
TYPE     Elevator   Assign x x+100

TYPE     Request    Box window 500 3+floor*100 54 44 "black" 1 "green"
```

(b)



(c)

Figure 7: *Usage of custom visualization techniques as a debugging tool. In (a) a partial listing of the elevator simulator is shown. A building consists of three elevators, and one request object to model pending requests. In (b) a visualization script is shown to visualize the running program. (c) shows one particular snapshot of the animation. A request is pending to have an elevator come to the fourth floor. The animation shows a clear flaw in the algorithm. Elevator two is coming, while elevator one and three are much closer by.*

```
    List(List *, int);
};
```

Observe the following script:

```
1.  GLOBAL Sort     Window 400 400 600 200
2.  LOCAL  List x   Integer 570
3.  TYPE   List     Box Sort x 50 9 value "black" 1 "gray80"
4.  ENSURE List     x < next.x-15
```

When this script is interpreted, Line 1 results in a window being created to show the animation. Line 2 adds an instance variable to the class definition of List. This is a symbolic addition, and the variable can be used later on in the visualization, as if it were an instance variable defined in the program. At line 3, a view is defined for `List` instances. This line defines that each instance of the type `List` should be represented by a `Box`. The position of the box is related to the actual value of the `x` field, which is defined at line 2. The width of the box is constant. Its height is a function of the `value` field, which is defined in the application (see the definition). At line 4, a constraint is defined, ensuring that each `Box` representing a `List` instance is always to the left of the instance referred to by the `next` field.

When this script is run in a custom visualization session, the result is as in Figure 8.

If the definition of the visualization is to be done visually, an approach should be found to define both the mappings and the constraints visually, using the mouse (as done in [Borning 86]). Doing the mapping visually has many advantages, such as ease of use. Also, all constraints are visible at all time, and can be understood by looking at the picture. However, in visual languages addressing components by *name* (a common mechanism in textual programming languages), is very difficult. In the design of HotWire, it was decided to use a textual format for the scripting language, for the time being. Also, exchanging textual scripts for visualizations of libraries, for instance, is much simpler.

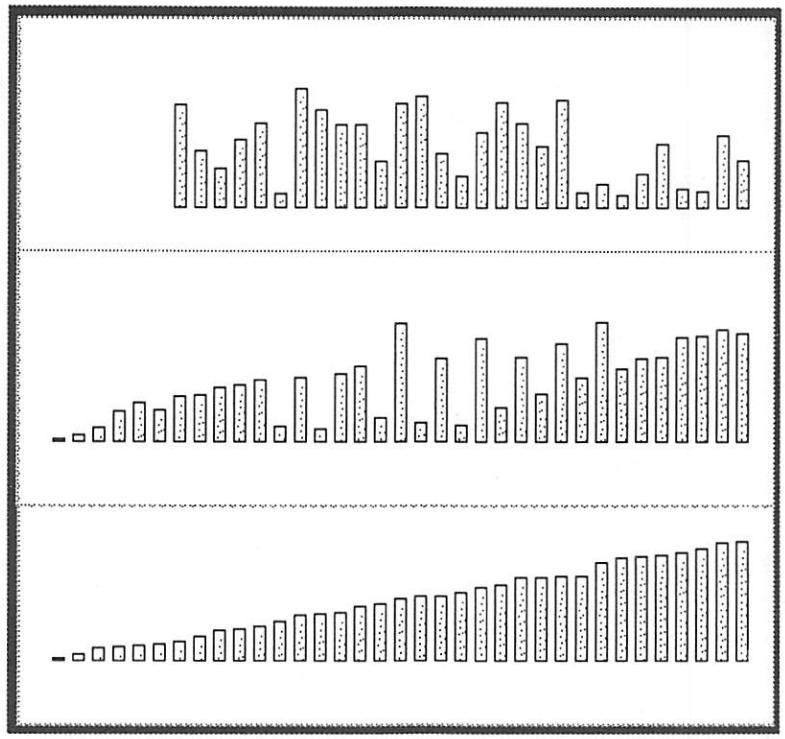


Figure 8: *A HotWire custom visualization of a sorting algorithm. The script given in the text of this paper will result in this animation of the algorithm. The top portion shows a snapshot during initialization of the datastructure. The middle part is taken during the execution of the algorithm, and the bottom view shows the state of the program, after it is finished.*

USING VISUAL TECHNIQUES FOR DEBUGGING

The foremost important result of using visual techniques for debugging is the enhanced understanding of the program. When one understands better how a program works, bugs or anomalies can be detected and identified much easier. Examples were given before how visualizations can address memory leaks, performance problems, incorrect execution threads, and flaws in the algorithm.

The visualization in the debugger can be static (created by the developers of the debugger), or dynamic (custom designed by the end-user of the debugger). In the first case, the debugger developer is striving to please a wide subset of programmers. Typically, a large number of different views of the executing program are supplied. The programmer will perhaps not use all of them, but specific problems may require different views highlighting particular aspects of the information that can be extracted from a running application.

An example is the approach taken it in the Procol runtime system [van den Bos and Laffra 89]. This started by showing the `MAIN` object (see Figure 9). Each Procol program contains one `MAIN` instance. Then, the mouse can be used to open up a browser for any instance shown on the screen, by clicking on it. When the browser is open, it can be dragged around the debugger window, and individual components defined in the instance, and shown by the browser, can be inspected into further detail, again by clicking on them. Each method that is activated on an instance, is animated by moving the cursor around (see the black rectangle in Figure 9). Also, at the end of a method, the values of the instance variables were checked, to see whether the display in the browser should be updated.

The inspection mechanism offered by this runtime system is useful for identifying a number of problems. However, the biggest drawback is that one first has to *find* the object that caused a specific problem. The instances shown in Figure 9 are the only ones that are shown, and most of the activity escapes our attention. In fact, we suffer from tunnel vision here.

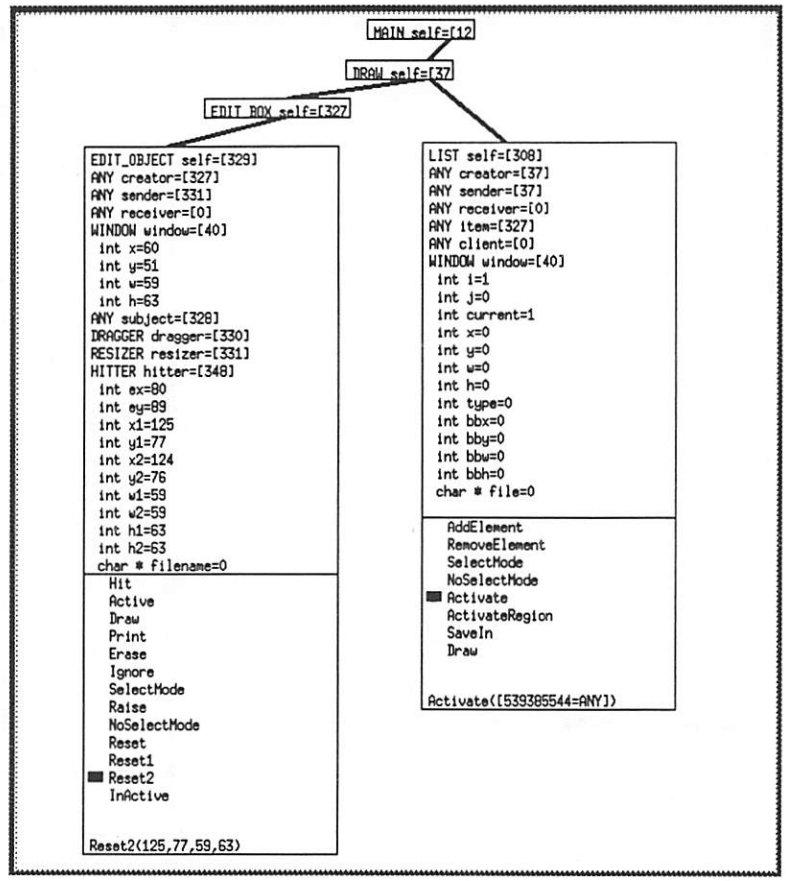


Figure 9: *This unprecedented, but still unsatisfactory view is produced by a previous attempt, the runtime debugger for the Procol language [van den Bos and Laffra 89]. It shows a couple of object browsers. Each of the browsers can be opened and closed. Shown are three closed browsers; for a* MAIN *object, a* DRAW *object, and an* EDIT_BOX *object. When a browser is open, individual elements can be clicked on, to create a browser for that specific instance too. Shown are the individual methods, and their activation is animated by moving the black rectangle to the currently active method. At the bottom, the name of the method last executed on this instance is listed, together with its arguments.*

Figure 10 shows a typical debugging session with HotWire. The program under study is *doc*, a word processor written in using the Interviews toolkit [Calder and Linton 91]. Figure 1 shows a snapshot of a HotWire debugging session of doc. First, a possible bug is observed. Namely, doc has a facility that one can create floating figures. After the figure has been created, it can be moved around on a given page. While doing that, our observation was that the figure moves fast in the bottom right corner, but moves slower and slower once we get closer to the top left corner. We experimented with different figures, and with different underlying documents. The effect remained the same. Our initial step towards the solution of this problem, was to look at the regular views that HotWire has to offer. We had HotWire show all objects and all method invocations. We tried the call stack view with the recording strip. But, none of the views really gave different results depending on the location of the figure. What we did find out was that a lot is happening already to draw a flashing cursor (the class InsertMarker in doc). Furthermore, even though there is always only one InsertMarker active in a doc session, a lot of others may also be invoked. Looking into more detail into the class InsertMarker (by clicking on the type name label in the instance window), we could see all methods that were invoked on all instances of type InsertMarker. Apparently, the sequence was that one InsertMarker is told to flash(), and as a result, it and a number of other InsertMarkers are told to draw() themselves. This observation leads to our first assumption: When the bug occurs, too much is drawn.

Reasoning that redrawing has all to go through one single object, the type Canvas, it led us to investigate the

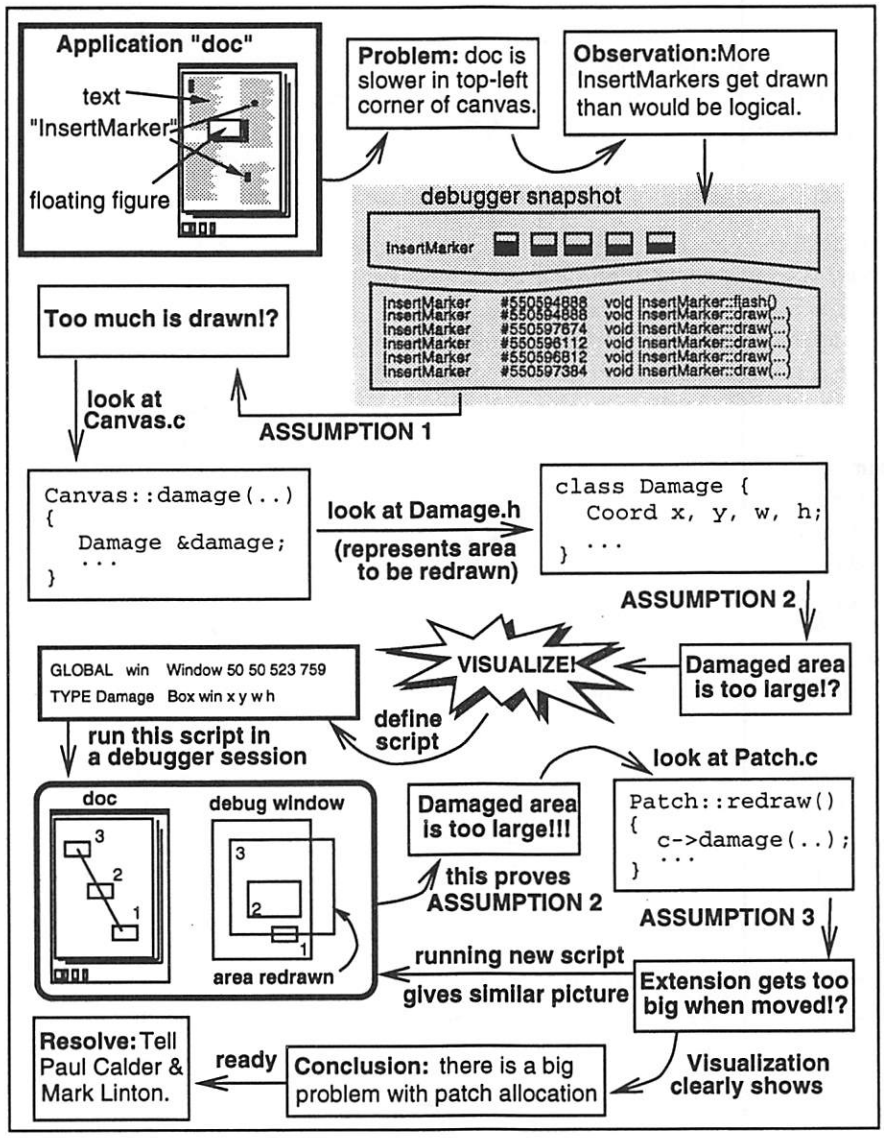


Figure 10: *A (busy) representation of the scenario for a typical debugging session with HotWire. Regular views are used to find anomalies, and customized visualizations are used to test and prove assumptions.*

damage() method. This method is the routine that is responsible for repairing damaged areas on the screen. Looking at the implementation of the method, we observed that an object of the type Damage is created for each repair operation. Sometimes, this object is saved from one operation to the next, to merge two areas that are overlapping into one new area. The Damage object will store the area that is to be repaired. It seems logical to test our assumption that too much is drawn, by inspecting the values of the instance variables of this Damage object. When we would be using an old-fashioned debugger, we would set a breakpoint in routine Damage::damage(), and inspect the values of the parameters. However, that really would not be practical, because we would stop too many times, as redrawing happens really often. Furthermore, the standard views of HotWire do not give any extra help here too. What we want to do next is to actually highlight the area that is redrawn inside doc. To do it in a general fashion directly in the doc canvas would be difficult to do, as it would make to many assumptions on the implementation of the application. Therefore, we will use a separate window of the same size of the canvas and visualize the area that is being redrawn in it.

In order to visualize the area that is damaged in the canvas, we define a small script. Basically, we tell HotWire to create a window, and to create a Box object (which is part of our scripting language) for each instance of the type Damage that is created during our debugging session. The size of the box is determined by the instance variables x, y, w, and h. These variables are C++ member fields, defined in the class Damage. Then, we run the application again, and observe what is happening. Now, we will see exactly which area is being redrawn. During initialization we ignore the flashing, but look more closely when we create a figure, and drag it around. When we start at the right bottom corner, the area that is repaired is roughly the same size as the figure itself. Occasionally, we move the mouse a little bit faster, and two areas are merged, so that the area that is repaired is somewhat larger. Which is ok. However, the more closely we move the mouse to the left top corner, the larger the damaged area gets! In fact, the are covers the entire window when we finally reach the top left corner. Experimentation learns us that the center of the canvas is the point of change. Looking at how a figure is implemented, we found out that it is using the class Patch, which is related to the Extension class. Visualizing that class in a similar way slowed down the visualization tremendously, as there happen to be a lot of extensions around, but in the end gave the same result. All this lead us to the conclusion that the patch allocation algorithm in the doc implementation has a serious problem.

As an aside, we learned a lot more from our visualization session of the damaged area. We found out that typing text in an empty window causes the entire line to be redrawn (until the right border of the page), even when there is no text after it. We did not consider that to be too serious, but still one might optimize that. What was far more serious, was the result of hitting the Enter key on the keyboard to go to the next line in our text. The result was a redraw of the entire page. Not one time, but exactly seven times. Visualizing the Insertmarker at the same time, learned us that the InsertMarker gets resized to be as high as the entire page, moves to the next line, and is then resized to the original line size again.

Summarizing, visualization can lead to interesting observations, and may highlight problems that one is not aware of, or that would be very hard to find using printf statements, tracing facilities, or existing debuggers.

IMPLEMENTATION

HotWire was initially developed on an RS/6000 running AIX with X Windows. Both the target and implementation languages were C++ and SmallTalk. The implementation is separated into two major parts. One part is the visualization code (approx. 4000 lines of C++). That includes all visualizations described in this paper, and also a parser and interpreter for the declarative scripting language. The other part is a graphical toolkit, with support for windows, structured graphics, display lists, double buffering, etc. (approx. 3000 lines of C++). The parser for C++ is developed using Lex and Yacc (approx. 1400 lines). An OS/2 port of the graphics toolkit is currently being worked on. No part of the functionality of HotWire needs to be changed to enable the porting. The reason for developing a new graphics toolkit, instead of using an available one, is the fact that efficiency was more important in our case than generality.

When a program is to be investigated, using HotWire, the C++ code is first annotated with a special pre-processor. This pre-processor reads in the code, instruments it at special locations, and feeds it into a standard

C++ compiler. The user does not have to change anything in the original source code. The instrumentation process is entirely automatic. The instrumented code makes calls to a runtime library, which is described before. It is essential that the visualization software runs in the same address space as the program, to reduce the overhead of the visualizations. HotWire behaves more like an intelligent runtime system, than as a debugger running in a separate process. The entire architecture is independent of the C++ compiler used. Furthermore, the interface to the graphics system is very small (one single class does all the graphics calls), all leading to a highly portable visualization tool.

CONCLUSIONS

C++ is being used for numerous program development projects for almost a decade already. Surprisingly, the state of the art in *debuggers* for C++ is still a few generations behind, and deserves a lot of attention. C++ debuggers use wrong metaphors, and especially the interface of most current debuggers is archaic, and needs to be replaced by one that uses more recent techniques. One promising approach is to use visualizations. They dramatically improve the quality of debuggers.

This paper describes **HotWire**, an experimental visual debugger for C++ and SmallTalk, developed at IBM T.J. Watson Research Center. It uses different types of visual techniques to enhance the understanding of the program under study. The main idea is to provide the user (the programmer debugging an OO system), with as many different views as required to help finding a bug. The visualizations are static (defined by the developers of HotWire), and custom built (defined by the end-user). The static visualizations emphasize on showing all classes and their instances, highlighting interactions on individual instances, showing method calls, recording all messages that happened in a particular time frame – and replaying and inspecting them afterwards, displaying the values of instance variables in a continuous manner, and showing relationships between instances. The custom visualizations are to be defined using a special visual scripting language. The script identifies the *models* – the objects of interest in the application –, the *views* – suitable shapes for displaying the objects –, and a *controller* – serving as a mapping between the two. The controller is declarative (an important aspect), and defines the views by referring to instance variables in the models. When the models get manipulated, the corresponding views will be updated by the visual debugger. In addition, triggers can be set on entering and leaving of methods, so that the visualization can be altered (animated), depending on activations of instances. Relationships between different views can be maintained using constraints.

There are problems that still need to be solved in the realm of visual debugging. One problem is how to display many classes and many instances. Appropriate abstraction mechanisms (as used in [Honda and Yonezawa 88]) and browsing techniques (like fisheye views [Sarkar and Brown 92]) need to be investigated to handle these types of complex (and realistic) systems. Furthermore, visualizations for libraries of source code could be developed. It is quite conceivable that toolkits of classes can be equipped with standard visualizations. For each part of the toolkit a special script could be defined, and be shipped with the toolkit or the debugger. The application of toolkits could be improved a lot, by educating programmers by showing visualizations. Finally, the structure of the script language is still being studied. The current version is the first generation. The aim is to keep the language declarative. Keeping the script declarative, reduces the complexity of the language and keeps the scripts small and concise, making the development of visualizations of C++ programs a simple and affordable task.

References

[Borning 86] Alan Borning, Defining Constraints Graphically, ACM–CHI'86 Conference Proceedings, April 1986, pp. 137-143.

[Brown 88] Marc Brown, Exploring algorithms using Balsa-II, IEEE Computer, 21, 5, May, 19988, pp. 14-36.

[Calder and Linton 91] Paul Calder and Mark Linton, Glyphs : Flyweight Objects for User Interfaces, UIST'91 Conference Proceedings, 1991.

[De Pauw 93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides, Visualizing the Behavior of Object-Oriented Systems, OOPSLA '93 Proceedings, ACM SIGPLAN Notices, October 1993, pp. 326-337.

[Freeman-Benson 90] Bjorn Freeman-Benson, Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming, ECOOP–OOPSLA'90 Conference Proceedings, Ottawa 1990, pp. 77-88.

[Goldberg 83] Adele Goldberg, *Smalltalk-80, The Interactive Programming Environment*, Reading, Addison-Wesley, 1983.

[Honda and Yonezawa 88] Yasuaki Honda and Akinori Yonezawa, "Debugging Concurrent Systems Based on Object Groups", In Proceedings of *ECOOP '88*, pp. 267–286, August 1988.

[Johnson 93] Ralph Johnson (compiled the list), "Classic Smalltalk Bugs", stored at host `st.cs.uiuc.edu` in file `pub/st-docs/classic-bugs`, accessible through the internet, 1993.

[Leler 88] William Leler, Constraint Programming Languages – Their Specification and Generation, Addison Wesley, Reading Mass., 1988.

[Meyer 88] B. Meyer, *Eiffel – The Language*, Prentice Hall, London, 1992.

[Miya 93] Eugene N. Miya, *comp.graphics.visualization FAQ*, UseNet, 1993.

[Sanella 93] Michael Sanella, John Maloney, Bjorn Freeman-Benson, and Alan Borning, "Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm", Software – Practices and Experiences, 25, 5, pp. 529–566, May, 1993.

[Sarkar and Brown 92] Manojit Sarkar and Marc Brown, "Graphical Fisheye Views of Graphs", in Proceedings of *CHI '92*, pp. 83–91, May, 1992.

[Smith 87] Randall Smith, Experiences with the Alternate Reality Kit – An Example of the Tension Between Literalism and Magic. CHI+GI'87 Conference Proceedings, April 1987, pp. 61-68.

[Stasko 90] John Stasko, TANGO: A Framework and System for algorithm animation, IEEE Computer, 23, 9, September 1990, pp. 27-39.

[van den Bos and Laffra 89] J. van den Bos and C. Laffra, "PROCOL - A parallel object language with protocols", In *ACM–OOPSLA'89 Conference Proceedings*, New Orleans, Special Issue *SigPlan Notices*, 24, 10, pp. 95–102, October 1989.

A Customisable Memory Management Framework

Giuseppe Attardi *
Tito Flagella †

Abstract

Memory management is a critical issue for many large object-oriented applications, but in C++ only explicit memory reclamation through the `delete` operator is generally available. We analyse different possibilities for memory management in C++ and present a dynamic memory management framework which can be customised to the need of specific applications. The framework allows full integration and coexistence of different memory management techniques. The Customisable Memory Management (CMM) is based on a *primary collector* which exploits an evolution of Bartlett's mostly copying garbage collector. Specialised collectors can be built for separate memory heaps. A `Heap` class encapsulates the allocation strategy for each heap. We show how to emulate different garbage collection styles or user-specific memory management techniques. The CMM is implemented in C++ without any special support in the language or the compiler. The techniques used in the CMM are general enough to be applicable also to other languages.

1 Introduction

As an alternative to explicit reclamation of heap memory, automatic recovery of unused memory can be performed through the technique of *garbage collection*. The garbage collector's function is to find data objects that are no longer in use and make their space available for further use by the program. An object is considered *garbage*, and therefore subject to reclamation, if it is not reachable by the program via any path of pointer traversal. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never follow a "dangling pointer" leading to a deallocated object.

There are good reasons to prefer automatic memory management: *safety*, avoiding the risk of deallocating an object too soon; *accuracy*, avoiding to forget to deallocate unused memory; *simplicity*, assuming a computational model with unlimited memory; *modularity*, the program does not have to be interspersed with bookkeeping code not related to the application; *reduced burden* on programmers who are relieved from taking care of memory management. Nevertheless garbage collection has not yet come into general use, sometimes for fears of losing efficiency but mostly for the lack of availability of the technique.

Recent research has proved that many of the limitations of traditional garbage collection techniques can be alleviated. Some experiments have even shown that explicit memory deallocation (using primitives like `free` or `delete`) is not necessarily faster than automatic reclamation of free memory

The research described here has been funded in part by the ESPRIT Basic Research Action, project PoSSo. Part of this work has been done while the first author was visiting the International Computer Science Institute, Berkeley, CA.

*Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy. Net: `attardi@di.unipi.it`.
†Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy. Net: `tito@di.unipi.it`.

[Breuel 92]. Techniques like *generational* garbage collection have been developed to minimise latency during garbage collection.

While these experiences have proved that garbage collection is a valuable technique, the variety of proposals is in itself an indication that the ideal garbage collector is impossible to achieve. A good design is one that strikes the appropriate tradeoff among many conflicting goals.

We faced the task of developing memory management facilities for a large research project: the ESPRIT BRA PoSSo aims at building a sophisticate symbolic algebra system for solving polynomial systems. Researchers working on different parts of the system have different requirements on the memory management. Some users prefer a copying garbage collector in order to maintain locality of their data. Others prefer a mark-and-sweep approach because of the fixed size of their data. The core algorithms of PoSSo required a special kind of memory management due to the particular FIFO dynamics of memory usage exhibited in certain portions of the Buchberger algorithm for computing a Gröbner basis [Buchberger 85].

These requirements led us to design a framework whereby users can select among different garbage collection strategies, ranging from manual management to fully automatic garbage collection, and can also implement their own specialised memory management as appropriate for their task. Without the support provided by our framework, if memory management were left to each programmer:

1. each user would have to introduce support for memory management in his code. This means adding extra fields to data and providing code for basic memory management operations, like computing the size of objects, the address ofthe next object in the heap, etc.

2. in large applications where different memory management facilities are required, different interfaces would be present for each memory manager (MM).

3. it would be impossible to mix data under different MM's. If an object under control of one MM contains a reference to an object controlled by another MM, such reference may not be noticed by the first MM, leading to incorrect memory reclamation.

When an intensive use of such facilities is required and a variety of memory management techniques are needed, the programs become very difficult to write and maintain, and subtle "memory leak" bugs may arise which are nearly impossible to find.

The Customisable Memory Management (CMM) addresses these issues by providing a general framework within which several policies can coexist. The framework takes full advantage of the object oriented paradigm of C++, and provides a consistent and simple interface for programmers.

The CMM is a memory management facility supporting complex memory intensive applications in C++. It consists of:

1. a general purpose garbage collector for C++; this collector is called *primary garbage collector* and is a variant of Bartlett's mostly copying collector [Bartlett 89];

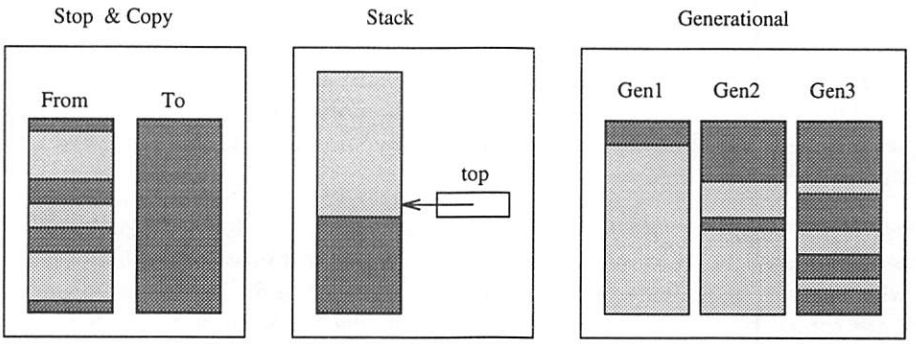
2. a user interface: this is the interface used by programmers to access the CMM;

3. a programmer interface: a set of facilities used by CMM programmers to define specific memory management policies as appropriate for their applications.

With CMM users can select among several predefined memory management facilities, define their own, or customise those provided in the framework.

For instance it is conceivable a situation like in the following figure, where three different memory management policies are available or even used together in the same application: a traditional stop-and-copy collector, a specialised stack allocator for portions of the algorithm with controlled

behaviour or a generational collector for real-time tasks such as user interfaces. The first two mechanisms are already available in the CMM, while a generational collector has been implemented by Bartlett [Bartlett 89].



The mechanism to implement these alternative policies is the Heap abstraction which we develop in this paper. Specific algorithms are used and particular data structures are maintained by each Heap to ensure its proper behaviour. A critical question is what to do with pointers which cross the boundaries of Heaps. If no such pointers are allowed, then a Heap need only to be concerned with objects it has allocated and on which it has some control. We considered this solution too restrictive, since it would not allow portions of applications built separetely by different people to freely exchange data. We took therefore special care to design the mechanism of Heaps to ensure that different Heaps can coexist and data of different sources can be mixed. The amount of coordination necessary to achieve this goal, has been kept to a minimum, and consists of a few simple functions that each class of collectable objects must provide and which it would be possible to generate automatically. To achieve coordination in a simple and effective way, we exploit the object oriented features of C++. In practice, all the operations of the collector are performed through member functions of the class of each object. However, the action of the collector on an object may vary also depending on the Heap where the collection started, not just on the Heap to which the object belongs. For instance if the collection starts in the Stop&Copy Heap, it applies its methods to mark and traverse the object in that zone, but if a pointer leads into a StackZone, those objects are unobtrusively traversed without modifying them. Only if such traversal leads back into the original Heap, the full collector operation resumes.

2 Requirements

In designing the CMM we tried to achieve the following goals:

- *portability*: the CMM is simply a library of C procedures and C++ classes, which can be used with any C++ compiler. Alternative solutions rely on changes to the underlying language or compiler.
- *coexistance*: code and objects built with the CMM can be exchanged with traditional code and libraries. No restriction exist on whether a collected object can point to a non collected object and viceversa. We wanted to be able to pass collected objects to programs unaware of garbage collection, allowing them to store such objects in data structures, without special burden on the programmer or risk that the object would be garbage collected. Alternative solutions require the programmer to put an object in an "escape list" before passing it to an external procedure.

- *algorithm specific customisation*: the allocation policy can be customised to the particular needs of an algorithm. This is different from other solutions, where the allocation policy is associated to the type of an object. For instance, in the proposal by Ellis and Detlefs [Ellis 93], it is possible to specify whether an instance or a class is allocated in the collected heap, rather than in the non collected heap. For the purpose of our applications, it is necessary to allocate the same type of object sometimes with one policy and sometimes with another. For example, in PoSSo there is only one class of polynomials, but sometimes a polynomial is allocated in a zone which can be freed quickly once a certain portion of the simplification algorithm is complete; some other time the lifetime of the polynomial cannot be predicted, so it must be allocated in the general heap zone.

- *multiple logical heaps*: at least two heaps are necessary, one for collectable objects and one for traditional objects. However two is not enough: for instance collectable objects containing data which cannot be relocated for some reasons must be handled differently from other objects which are copied by the collector. For this reason the CMM provides multiple logical heaps, called Heap.

- *usability*: only minimal burden is placed on the programmer who wants to use the collector. When collectable objects are required the programmer needs to define their class as inheriting from the base class `GcObject` and supply a method for traversing them, a task which could be automated.

- *separation of concerns*: memory management code needs not to be included within algorithms, and it is possibile to change the memory policy just by selecting which heap is employed by the algorithm.

- *efficiency*: the implementation is efficient enough to be as good and sometimes better that hand tuned allocation.

The CMM allows customisation of the collector and provides a few pre-built variants. One could argue whether a single general strategy could fit all the needs. For instance a generational garbage collector ensures that memory is reclaimed quickly. However not even a generational garbage collection is good enough for applications like PoSSo where one must prevent or delay garbage collection as much as possible, not just make its duration shorter. For the vast majority of applications a general purpose strategy is adequate, and the CMM provides a good one by default. But for research or applications that need to push the limits of technology, the CMM provides a solution with limited effort on the user.

In the rest of the paper, we recall the general principles of memory management, then present our primary collection algorithm, then discuss the CMM, its implementation and its usage. Finally we illustrate how to emulate different garbage collector styles and application specific memory management policies.

3 Dynamic Memory Management: Concepts and Terminology

A garbage collector in principle could reclaim the space occupied by all objects that the running program will no longer access. Unfortunately this is an undecidable property; therefore garbage collectors must adopt a simpler criterion based on the notion of potentially accessible or *live object*. A garbage collection mechanism basically consists of two parts [Wilson 92]:

1. distinguishing the *live objects* from the garbage in some way, or *garbage detection*;

2. reclaiming the garbage objects' storage, so that the running program can reuse it.

The formal *criterion* to identify *live* objects is expressed in terms of a *root set* and *reachability* from these roots. The *root set* consists of the global and local variables, and any registers used by active procedures. Heap objects directly reachable from any of these variables can potentially be accessed by the running program, so they are *live* objects which must be preserved. In addition, since the program might traverse pointers from these objects to reach other objects, any object reachable from a live object is also live. Thus the *live set* is the set of objects in some way reachable from the roots. Any object not in the live set is garbage and can be safely reclaimed.

Several variations are possible on this general working schema, depending on:

1. how to identify the roots (conservative, explicit registration, smart pointers, etc);
2. how to identify internal pointers pointing to other GC objects (compiler support, user support, conservative, etc);
3. how the GC distinguishes live objects from garbage (marking or copying, with their many variants).

Quite different implementations result from different combinations of the above techniques. We can characterise as follows, according to these criteria, some of the most recent implementations of garbage collectors for C++:

	Identify roots	**Identify internal pointers**	**Distinguish live objects**
Boehm	Conservative	Conservative	Mark
Edelson	Smart Pointers	User assisted	Copying
Bartlett	Conservative	User assisted	Promotion & Copying

Depending on the kind of information available during the traversal of objects from the root set, a tracing collector can be *conservative*, *type-accurate* or both.

A *conservative* garbage collector does not require cooperation from the compiler and assumes that anything that *might* be a pointer actually *is* a pointer. In this case an integer (or any other value) is assumed to be a pointer by the collector if it corresponds to an address inside the current heap range: any such value is called an *ambiguous root*. A garbage collector is *type-accurate* when it is able to distinguish which values are genuine pointers to objects. Some garbage collectors adopt a combination of these two techniques: some pointers are dealt conservatively, while others are treated in a type accurate way.

The main limitations of a purely conservative collector are memory fragmentation in applications handling objects of many different sizes, arising from the inability to move objects, and the risk that a significant amount of memory might not be reclaimed in applications with densely populated address spaces of strongly connected objects [Wentworth 90].

The alternative approach which is *type-accurate* in identifying objects faces some non trivial problems with hidden pointers. One such case is the `this` pointer in C++: whenever a method is invoked on an object, a pointer to that object is passed to the method via the stack as the implicit local variable `this`, but only the compiler knows where such variable is actually located. The only compiler-independent way to catch such pointers is to examine the stack conservatively. Failing to trace this

pointer is dangerous: the object might be reclaimed or moved without updating the pointer. In both cases a *dangling pointer* is generated with serious consequences for the integrity of the program.

Both these limitations are avoided in the partially conservative approach proposed by Bartlett for his *mostly copying garbage collector*. We chose this technique as the basis for developing our customisable collector.

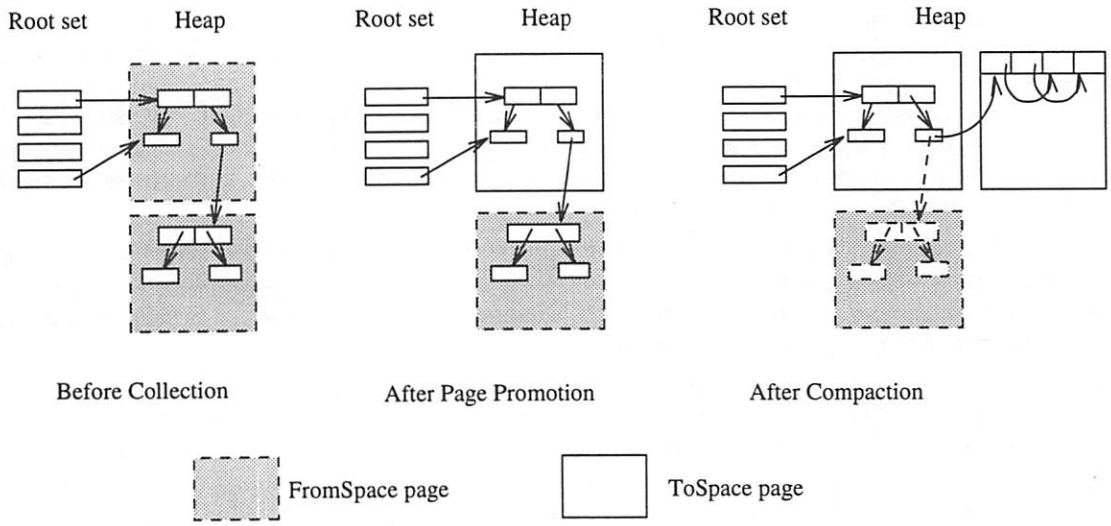
4 The Primary Collector

The Customisable Memory Management relies on an underlying general mechanisms for identifying objects, moving them and recovering memory. These mechanisms constitute the *primary collector* of the CMM and are based on Bartlett's technique. We illustrate here the technique and how we improved it for our needs.

4.1 Bartlett's mostly copying collector

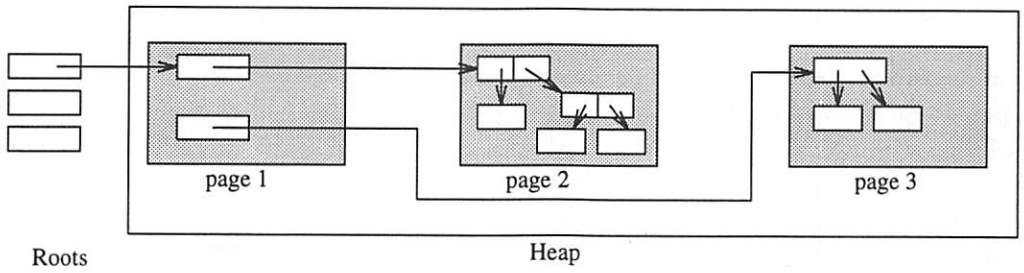
A mostly-copying garbage collector performs compacting collection in the presence of ambiguous pointers in the root set. Bartlett's implementation (BGC) is an evolution of the classical stop-and-copy collector which combines copying and *conservative* collection. BGC does not copy those objects which are referenced by ambiguous roots, while most other live objects are copied.

The heap used by BGC is a non necessarily contiguous region of storage, divided into a number of equal size pages, each with its own *space-identifier* (either *From* or *To* in the simplest non generational version). The *FromSpace* consists of all pages whose identifier is *From*, and similarly for *ToSpace*. The collector conservatively scans the stack and global variables looking for potential pointers. Objects referenced by ambiguous roots are not copied, while most other live objects are copied. If an object is referenced from a root, it must be scavenged to survive collection. Since the object cannot be moved, the whole page to which it belongs is saved. This is done by *promoting* the page into *ToSpace* by simply changing its page space-identifier to *To*. At the end of this promoting phase, all objects belonging to pages in *FromSpace* can be copied and compacted into new pages belonging to *ToSpace*. Root reachable objects are traversed with the help of information provided by the application programmer: the programmer is required to add a few simple declarations which enable the collector to locate the internal pointers within objects.



4.2 Revised Algorithm

Experimenting with the original implementation of Bartlett's mostly copying algorithm, we noticed that for some of our programs the amount of garbage not reclaimed was too high. The main reason for this was that a whole page was promoted when it contained just a single object reachable from a root: all objects in that same page will be preserved as well as their descendants, thereby missing to reclaim significant amounts of memory. This is illustrated in the following figure, where the object in the rightmost heap page is retained since it is pointed from within the leftmost page which has been promoted.



To improve the ability to reclaim storage of Bartlett's algorithm we keep a record of those objects actually reachable within a page being promoted during the first phase. This allows us to identify reachable objects in promoted pages. This information is contained in a bit table called **`LiveMap`**.

Here is our revised version of Bartlett's algorithm, which in most cases is still iterative:

1. Clear the **`LiveMap`** bitmap
2. Scan the root set to determine objects which cannot be moved. Any directly reachable object is marked as *live* setting a bit in the **`LiveMap`** bitmap and the page to which it belongs is promoted.
3. Scan each promoted page linearly, looking for live objects. Traverse each live object by applying the following procedure to each pointer it contains:
   (a) if the pointer lays outside the heap do nothing;
   (b) if it points to an object not yet reached: scavenge the object if it is in a promoted page, i.e. copy it, mark the copy as *live*, set a forwarding pointer within the object to the copy. Otherwise mark the object *live* and, in case it is past the current scanning position, recursively traverse it.
   (c) if it points to a *live* object in a non promoted page update the pointer to the forward position.

All new pages allocated for copying reachable objects belong to *ToSpace*, therefore the algorithm does not need to traverse copied objects. A copied object is traversed when the collector examines its page, so traversal is rarely recursive.

This algorithm does not require a forward bit as used in Bartlett's implementation: we can determine that an object has been forwarded if it is marked as live and contained in a non-promoted page. We also do not need to store in each object its size which Bartlett requires in order to scan through the objects in a promoted page. And finally, since we can determine the heap to which an object belong from its address, we can completely get rid of the one word of header required in Bartlett's algorithm, therefore eliminating any space overhead for collected objects.

Our experiments with the new algorithm show improvements up to 50% in the amount of space reclaimed with the new algorithm.

5 Multiple Heaps

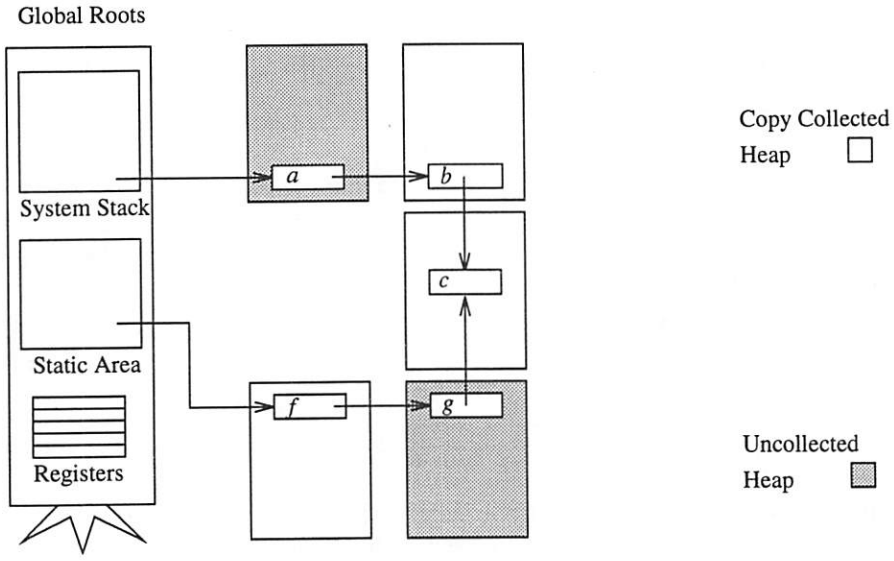
Bartlett's algorithm creates and manages a heap of objects which are collected by copying. The traditional uncollected heap is still available through the primitives `malloc` or `new` on uncollected classes. The uncollected heap cannot be eliminated since there are programs and libraries which may use uncollected object in an unsafe way for the collector [Ellis 93], and there are objects that can't be relocated. It must be possible however that objects in the uncollected heap point to objects in the collected heap and viceversa.

Pointers across heaps must be dealt carefully. The original Bartlett's implementation requires that pointers to collected objects be registered as roots. This is not practical, since it would entail registering as root any collected object which is passed to a library which might store it internally. This can be cumbersome to do and may be accidentally forgotten.

Therefore we need to extend the collector algorithm so that it is capable of discovering such pointers. The solution will later be generalized to deal with other logical heaps, created and maintained by users.

The uncollected heap should be considered as part of the root set. An obvious solution would then be to scan conservatively the entire uncollected heap searching for pointers to collected objects. This would be too expensive and would posit as live also objects pointed from unreachable locations in the heap. Alternatively one could perform a first complete scan from the root set to identify cross-pointers from uncollected to collected objects, in order to promote the pages where the latter reside, and then the mostly-copying algorithm would be applied. This is also a costly alternative, since it requires traversing twice the objects.

If we examine where Bartlett's algorithm fails, we can figure out an alternative solution. In the following figure, object c is pointed both from b, in the copy collected heap, and from g, in the uncolleted heap.



If we apply the mostly-copying algorithm, the pages where b and f will be promoted since they are pointed from roots. In the copy phase object c would be copied to a new page and the pointer in b will be updated. However, when later we reach c from g, we discover that its page should have been promoted. We could in fact promote it now, if only b had not been updated. This in fact suggests a solution: we do not update pointers when an object is copied, but we just record the location to be

updated, using a temporary bitmap. If we discover that the object should not have been moved, we restore all the objects in its page from their copies. The updates to pointers are performed only at the end of the algorithm, using the bitmap and the forwarding pointer stored in the objects. This technique is similar to the one suggested by Detlefs [Detlefs 92] to handle C/C++ unions of pointers and non-pointers.

5.1 User Collected Heaps

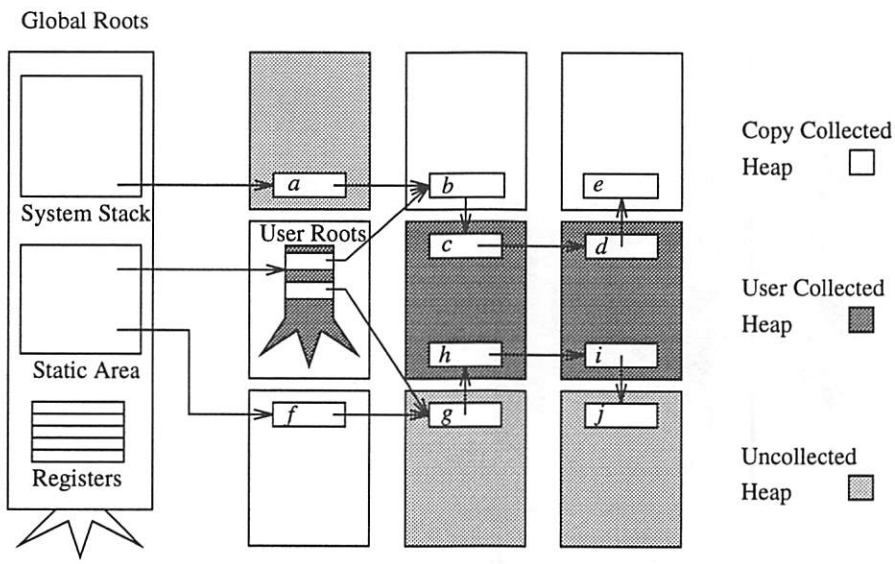
With the algorithm described so far, two heaps are available: an uncollected heap for non garbage collected objects and a collected heap.

Our goal is to allow users to build their own heaps with specific allocations strategies for their applications.

We must however fulfill some essential requirements for the solution to be consistent and practical:

- allow pointers across heaps: restricting the range of pointers is difficult and inconvenient.
- transitivity of liveness: if an object is pointed by a live object it is live as well. We must ensure that a pointer crossing heap boundaries does not go unnoticed by the collector.
- independence of collectors: it must be possible to write a collector for a particular heap, without relying on the collectors for other heaps, provided the root set for such heap is known.
- coordination among heaps: a simple set of conventions is established to ensure that pointers across heaps can be properly traversed.

In the following figure three heaps are present: the uncollected, the copy collected, and one user collected heap.



All six possible cross-heap pointers are shown. The user heap is maintained by the user, who keeps a record of the roots into his heap, so that he can perform a collection relative to that heap when appropriate, without involving the general collector. However the general collector must be capable of identifying for instance object *e* as live, even though this requires to cut across several heaps.

5.2 Customising the GC

The basic operations of a copying tracing collector are traversal and scavenging. The **traverse** procedure is used in the first phase of the collector to identify live object, the **scavenge** procedure is used to copy an object or perform whatever action is needed to preserve it.

Supporting multiple heaps requires to specialise these operations along two dimensions: according to the type of the object for traversal; and according to the heap where the object is located for scavenging.

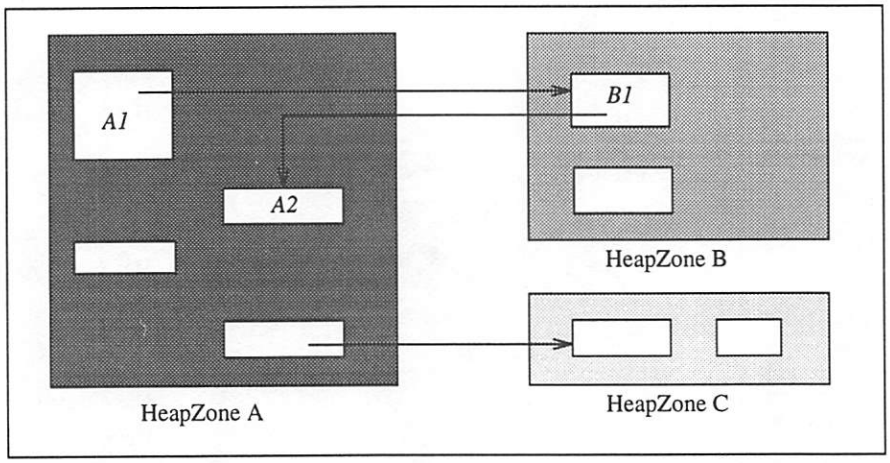
One way to customise these operations is to use the mechanism of *callbacks*, used for instance in programming window based user interfaces. With this schema, a user would register a specific callback routine with the general garbage collector, for use on specific type of objects. So when the garbage collector recognises one of these objects during traversal, it applies the appropriate callback to collect the object.

Callbacks can be different for each individual object, but this is not necessary for our purposes, so we preferred to replace callbacks with member functions. This makes these functions more convenient to define and to retrieve by the collector through the standard mechanism of C++. Moreover the **traverse** function could actually be generated automatically and no registration has to be added in the application programs. A version of our algorithm for C would still exploit callbacks.

5.3 Coordination

To achieve coordination among collectors for the various heaps, one has to agree to a mechanism that allows traversing objects in different heaps on behalf of the collector for another heap. While traversing a foreign heap, a collector should not be allowed to make changes to the objects it visits, except to recognized pointers to an object in his heap, when the object is moved.

So it is important that the traversal mechanism is uniform but capable to distinguish by whom it was initiated. This is achieved supplying a Heap parameter to **traverse** and making **scavenge** a member function of a **Heap**. Consider for instance the following situation:



Suppose a garbage collection is started in Heap A which uses a copy collector. While traversing object $A1$, the garbage collector identifies a pointer to the object $B1$, belonging to Heap B. Object $B1$ is scavenged by the **scavenge** function of the Heap A. This function recognizes object $B1$ as external to Heap A, so it does not copy the object, as it would if it were internal to the zone, but only traverses the object to determine whether further objects in Heap A can be reached from it.

The behaviour of **scavenge** changes again when object $A2$ is reached which belongs to Heap A. Applying the scavenge function of Heap A has the effect of copying object $A2$.

6 The CMM Run Time

Heap memory is divided into pages of equal size. The allocator for each **Heap** requests pages from the low level page allocator, where to allocate its objects. Each page is tagged with the Heap to which it belongs.

The CMM provides a **malloc** routine which uses such pages to allocate objects, implementing the traditional uncollected heap. **malloc** actually creates an instance of class **CmmObject**, which contains an array of the required size, and returns a pointer to such array, as expected by calling programs. This is in fact an *interior pointer* inside an object, and we exploit the ability of the CMM to map such pointers to their base. This allows us to traverse also **CmmObjects** by means of its member function **traverse**, defined as follows:

```
void CmmObject::traverse(Heap* heap) {
   for (int i = 0; i < this->size(); i++)
      promote_page(block[i]);
}
```

so that it promotes pages which are pointed from within the block. The only essential information that **CmmObject** must provide is the size of the block.

In all other collected heaps, the objects allocated are instances of class **GcObject** or its derivatives, which have their specialised version of **traverse**. No space overhead is present in **GcObject** except for what C++ must supply for the support of virtual functions.

A bitmap is used to deal with internal pointers to objects. Whenever a CMM object is created, the bit corresponding to its first word is set. Using this information, a pointer inside that object can be normalized to the beginning of the object, simply scanning the bitmap backward until the first set bit.

When an object has been moved, its first word is replaced by a forwarding pointer to the new object. As already mentioned, this happens only during garbage collection and the collector can determine this situation from the fact that the object is marked *live* and it is in a page in *FromSpace*.

6.1 The GcObject class

The run time support required for collectable objects is provided by the class **GcObject**. Every class of collectable objects is derived from **GcObject**.

Users access the services of the CMM mainly by using **GcObject** member functions. The most notable function of **GcObject** is the overloaded **new** operator which takes care of allocating the object in a specific heap. The other functions are used by the primary collector or by user defined collectors.

Here is the public interface for this class.

```
class GcObject
{
public:

  void* operator new(size_t, Heap* = (Heap *)heap);
```

```
  virtual void traverse(Heap* zone);

  GcObject *next();              // returns the next adjacent object
  int forwarded();               // tells whether the object has been forwarded
  void SetForward(GcObject *ptr); // sets the forwarding pointer
  GcObject *GetForward();        // returns the forward location of the object
  Heap *zone();                  // returns the zone to which the object belongs
  void mark();                   // marking primitives
  bool IsMarked();
  void SetLiveMap();
};
```

7 CMM User Interface

A collected class must derived from the class **GcObject** provided by the CMM. The default collector calls the method **traverse** on collected objects to identify their internal pointers to other objects. Users have to provide **traverse** methods for each class whose data members contain pointers. **traverse** must be defined according to well defined rules presented below, because it implements the interface between the CMM and user defined collected objects.

These rules ensure that superclasses or class objects contained in the class are correctly handled. The following example illustrates the rules, which are a generalisation of those in [Bartlett 89]. Suppose the following collected classes were defined:

```
class BigNum: public GcObject
{
  long data;
  BigNum *next;                          // Rule (a) applies here
  void traverse(Heap *zone);
}

class monomial: BigNum                   // Rule (c) applies here
{
  PowerProduct pp;                       // Rule (b) applies here
  void traverse(Heap *zone);
}
```

A **BigNum** stores in **next** a pointer to a collected object which needs to be scavenged, so **traverse** becomes:

```
void BigNum::traverse(Heap *zone)
{
  zone->scavenge(&next);                 // Applying rule (a)
}
```

Because **monomial** inherits from **BigNum**, the method **traverse** for this base class must be invoked; finally, since a **monomial** contains a **BigNum** in **pp**, this object must be traversed as well:

```
void monomial::traverse(Heap *zone)
{
  BigNum::traverse(zone);                // Appling rule (c)
```

```
  pp.traverse(zone);                    // Applying rule (b)
}
```

Finally, to deal with multiple base classes, we must identify the hidden pointer to the base class present inside an object. This cannot be done in a compiler independent way, so the CMM provides a macro `VirtualBase` which is compiler specific. For instance, its definition for the GNU C++ compiler is:

```
#define VirtualBase(A)  & (_vb$ # A)
```

In summary the rules are:

(a) for a class containing a pointer, say `class C { type *x; }`, the method `C::traverse` must contain `zone->scavenge(&x)`

(b) for a class containing an instance of a collected object, say `class C { GcClass x; }`, the method `C::traverse` must contain `x.traverse(zone)`

(c) for a class derived from another collected class, say `class C:` *GcClass* `{...}`, the method `C::traverse` must contain *GcClass*`::traverse(zone)`.

(d) for a class deriving from a virtual base class, say class `C: virtual` *GcClass* `{...}`, the method `C::traverse` must contain `zone->scavenge(VirtualBase(`*GcClass*`))`;

Preprocessing [Edelson 92] or compiler support [Samples 92] could be adopted to avoid hand coding of these functions and risks of subtle errors in programs. We plan to address this issue in the future.

7.1 Object Creation

When creating a collected object one can specify in which Heap to allocate it. The parameter `zone` can be supplied in the standard C++ placement syntax for the `new` operator:

```
p = new(zone) Person(name, age);
```

If the user does not specify any Heap, the default Heap `heap` is used:

```
p = new Person(name, age);
```

which is equivalent to:

```
p = new(heap) Person(name, age);
```

where `heap` is a global variable initialised to the system Heap.

When creating collected objects, the programmer can decide case by case where to allocate them. In summary, the following are the alternatives for object allocation:

Heap	Classes	Creation
uncollected	uncollected	`new` / `malloc`
copy collected	collected	`new`
user collected	collected	`new(zone)`

where we call collected those classes which inherit from `GcObject` and uncollected all others.

With the CMM, object allocation is not tied to the type of an object as in other proposals, so a programmer can design his classes without committing to a particular memory policy. The policy can be decided later, or even be different in different portions of an application. For instance, in the PoSSo solver, one sets the variable `heap` to the heap implementing the stack policy before starting the simplification. Throughout the simplification, all objects (monomial, polynomial, large precision integers, lists and so on) are allocated in this heap and freed in a single step at the end of the simplification. After simplification, one reverts to the normal heap. It is essential that this can be done without changing a single line in the user code.

8 Heap Classes

To manage a heap one normally has to maintain the set of roots for the objects in the heap, manage the pages where objects are allocated and implement the memory allocation and recovery primitives. A suitable encapsulation for these functionalities is provided by the `Heap` class.

8.1 The Heap Class

A class implementing a heap must supply definitions for the following pure virtual functions: `allocate` and `reclaim`, implementing the memory allocation strategy, `collect` to perform collection, and `scavenge`, the action required to preserve live objects encountered during traversal. Heap classes are derived from the abstract class `Heap`, defined as follows:

```
class Heap;
{
 public:
  int Index();                    // identifies the Heap

  Heap();                         // initializer

  virtual GcObject* allocate(int ObjSize) = 0;
  virtual void reclaim(GcObject* ObjPtr) = 0;
  virtual void scavenge(GcObject **ptr) = 0;
  virtual void collect() = 0;

  // Operations on the Root Set:
  void register(GcObject *);      // add an element
  void register(GcObject **);
  void deregister(GcObject *);    // remove an element
  void deregister(GcObject **);
  void ScanRoots(Heap *zone);     // scan the roots

  bool outside(GcObject *ptr);    // checks if pointer is outside this Heap
  void visit(GcObject *ptr) {
     if (! ptr->IsMarked()) {
        ptr->mark();
        ptr->traverse(this);
     }
  }
```

```
 private:
  int index;
  RootSet *roots;
};
```

`roots` is a pointer to an instance of class `RootSet`, used for registering potential roots. Depending on the particular type of `RootSet` used, the collector can be conservative, type-accurate or both. The simplest `RootSet` considers as possible roots only the objects explicitly registered by the user. The derived class `ConservativeRootSet` scans also the system stack, the process static data area, and registers for possible roots.

8.2 The Bartlett Heap

The Heap `Bartlett` encapsulates the primary collector of the CMM. The function `gcalloc`, `gcmove` and `gccollect` are the primitive functions provided by Bartlett's implementation of the collector.

```
class Bartlett: public Heap
{
 public:
  Bartlett() {
    roots = new ConservativeRootSet();
  }

  GcObject* allocate(int ObjSize) {
    return (GcObject *)gcalloc(GCBYTEStoWORDS(ObjSize));
  }

  void scavenge(GcObject **ptr) {
    if (OutsideHeap((int *)*ptr))
       return;
    GcObject *p = GetBeginning((int *)*ptr);
    if (outside(p))
       visit(p);
    else {
       *p->SetForward(gcmove(p));
       ToBeForwarded(ptr);
    }
  }

  void reclaim(GcObject* ObjPtr) {}; // delete does nothing

  void collect() {
    gccollect();                    // the actual Bartlett's collector
  }
}
```

Bartlett's collector starts scanning the set of possible roots to identify live objects. Because it is a conservative collector, `roots` is an instance of `ConservativeRootSet`. The objects it contains are traversed to identify other live objects. Objects are traversed in a type-accurate way by applying the user supplied function `traverse`. `traverse` applies in turn the Heap member function `scavenge`

to each reachable object. For each object in the Bartlett Heap, Bartlett's original `gcmove` primitive is used to copy it and compact memory; otherwise the object is visited using the function `visit`, which marks the object if necessary and then traverses it.

8.3 The Uncollected Heap

The uncollected heap is available through the default `new` operator or the functions of the `malloc` library. Objects not inheriting from `GcObject` are allocated in this heap.

8.4 The root set

Many heap zones require the user to explicitly register the possible roots. To support that, the class `Heap` contains an instance of the class `RootSet` supporting the following operations:

```
void set(GcObject *);
void unset(GcObject *);
void setp(GcObject **);
void unsetp(GcObject **);
```

`setp` and `unsetp` are used to (un)register pointers to GC objects as roots. `set` and `unset` are used to (un)register GC objects as roots. Consider the following example:

```
cell GlobalRoot;                                    // Define a cell variable

main()
{
  cell *LocalRoot = new cell;                       // Define a cell pointer
  HeapStack *MyHeap = new HeapStack(10000);         // Create a new heap zone
  MyHeap->roots.setp((GcObject **)&LocalRoot); // Register the pointer as a root
  MyHeap->roots.set(&GlobalRoot);                   // Register the cell as a root
  LocalRoot->next = new(MyHeap) cell;               // Allocates some new cells
  GlobalRoot.next = new(MyHeap) cell;
  MyHeap->collect();                                // The collector will identify
                                                    // any allocated cell, starting
                                                    // traversing from cell LocalRoot
                                                    // and GlobalRoot
  MyHeap->roots.unsetp((GcObject **)&LocalRoot); // Deregister the local root.
}
```

9 Implementing Heaps

This section illustrates the CMM programmer interface for implementing new Heaps. We describe the mechanism through an example, which is a simplified version of the actual Heap used in PoSSo.

9.1 The HeapStack

A foremost algorithm in the PoSSo algebra system is the one for computing of the Gröbner basis of a set of polynomials. Dependencies between temporaries and persistent data make the use of explicit memory allocation/deallocation nearly impossible, so use of a garbage collector was essential. The

main step of the Buchberger algorithm [Buchberger 85] consists in the simplification of a polynomial which involves many operations creating a lot of intermediate polynomials of which only the last one is relevant and is inserted into the basis. Once this polynomial has been computed, all the temporary structures allocated can be removed.

The peculiar dynamics of the problem offers an opportunity to try out the CMM facilities to implement a specific memory management. We created a Heap in which the allocation is stack-like (and thus fast), and the garbage collector called synchronously after each step.

We present a simplified solution in which the size of the stack is fixed, and a copying collector which uses two areas. The real solution we adopted for the problem is more complex and uses a list of areas, and a copying collector.

9.2 The HeapStack

First we define the `HeapStack` class as a `Heap` consisting of two areas which implement the `FromSpace` and the `ToSpace` of the collector, and a `RootSet` to register the roots to use for the collection:

```
class HeapStack: public Heap
{
  public:
   void scavenge(GcObject **ptr);
   GcObject* allocate(int words);
   void reclaim(GcObject* ObjPtr) {};
   void collect();
   HeapStack(int size = 100000);

  private:
   pages FromSpace, ToSpace;
   int FromTop, ToTop;
};

HeapStack::HeapStack(int StackSize)
{
   FromSpace = allocate_pages(StackSize, index);
   ToSpace = allocate_pages(StackSize, index);
}

inline GcObject* HeapStack::allocate(int size)
{
   int words = BYTEStoWORDS(size);
   int *object = FromSpace + FromTop;
   if (words <= (FromSize - FromTop)) {
      FromTop += words;
      return (GcObject *)object;
   }
   else return (GcObject *)NULL;
}
```

The collector uses the root set to traverse the roots using its traversing strategy. After having moved to `ToSpace` all the objects reachable from the roots, it traverses those objects in order to move all further reachable objects. The specific action required for scavenging objects is as follows:

```
void HeapStack::scavenge(GcObject **ptr)
{
    GcObject **OldPtr = ptr;

    if (OutsideHeap((int *)*ptr))
       return;
    GcObject *p = GetBeginning((int *)*ptr) ;
    if (outside(p))
       visit(p);
    else if (*ptr->forwarded())
       ToBeForwarded(ptr);
    else {
       *ptr = moveTo(ToSpace, *ptr);
       OldPtr->SetForward(*ptr);
    }
}
```

This code relies on support provided by classes **GcObject** and **HeapStack**. As the final step the collector exchanges the roles of **FromSpace** and **ToSpace**.

```
void HeapStack::collect()
{
   pages *TmpSpace;
   GcObject *ObjPtr;
   // First traverse the objects registered as roots, applying our scavenge
   ScanRoots(this);
   // Now traverse the objects already moved into ToSpace
   ObjPtr = ToSpace;
   while (ObjPtr < ToSpaceEnd) {
      ObjPtr->traverse(this);
      ObjPtr = ObjPtr->next();
   }
   // swap FromSpace and ToSpace
   TmpSpace = FromSpace; FromSpace = ToSpace; ToSpace = TmpSpace;
   FromTop = ToTop; ToTop = 0;
}
```

The **roots** for **HeapStack** can be set or deleted using the Heap member function register and unregister. In the case of the Buchberger algorithm we register two global variables containing the **Base** of polynomials and the list of polynomial pairs which are the only objects which need to be preserved after each simplification step:

```
HeapStack BBStack;
Base b;
Pairs p;
...
main() {
  BBStack.register(b);
  BBStack.register(p);
  ...
     BBStack.collect()
```

```
    ...
    BBStack.deregister(b);
    BBStack.deregister(p);
}
```

10 Related Work

The Boehm-Wiser collector [Boehm 88] is a well known collector for C++ which is convenient to use since it is totally conservative. However is not customisable and is subject to unduly retention of space and memory fragmentation since it cannot compact memory. Our copying collector has some advantage in performance not having to reconstruct a free list after collection and being more accurate in tracing live objects.

Work on adding garbage collection to C++ has been done by Dain Samples and Daniel Edelson. Samples [Samples 92] proposes modifying C++, to include a garbage collection environment as part of the language. This may be a good long term approach for garbage collection in C++ but is not suitable for a project like PoSSo which needs portable garbage collection facilities as soon as possible. Our feeling is that this work demonstrates how the flexibility of object oriented languages can be used to implement a very complex environment, like CMM, without requiring modifications to the language.

Edelson [Edelson 92] has been experimenting with the coexistance of different garbage collection techniques. The flexibility of the solutions he adopts in his approach allows the coexistance of different garbage collectors, but he does not provide any interface to the user to customise and/or define his own memory management facilities.

Ellis and Detlefs [Ellis 93] propose some extensions to the C++ language to allow for collectable object. The major change is the addition of the type specifier `gc` to specify which heap to use in allocating the object or a class. They also propose to change the operator `new T` to call the collector allocator when `T` is a `gc` type, and as a consequence of this, the overloading of `new` and `delete` operators for `gc` classes is forbidden. While the `gc` keyword is compatible with our solution of inheriting from the base class `GcObject`, the constraint on `new` needs to be relaxed to allow overloading of `new` when additional arguments are present. Otherwise this constraint will block the possibility of using different zones for the same kind of objects in different portions of a program. Other suggestions from the Ellis-Detlefs proposals are quite valuable, for instance making the compiler aware of the garbage collection presence and avoid producing code where a pointer to an object (which may be the last one) is overwritten. This can happen for instance in optimizing code for accessing structure members.

11 Conclusion

The CMM offers to programmers garbage collection facilities without significant compromises. They can use a generic collector, a specific collector or no collector at all, according to the need of each algorithm. The algorithm can be in control when necessary of its memory requirement and does not have to adapt to a fixed memory management policy.

The CMM is implemented as a C++ library, produced with extensive revisions from the original Bartlett's code. It is being heavily used in the implementation of high demanding computer algebra algorithms in the PoSSo project. The CMM provides the required flexibility without degradation in performance as compared to versions of the same algorithms performing manual allocation.

The next challenge would be to incorporate in the compiler the minimal facilities required for CMM support: the addition of the `gc` keyword, proposed by Ellis and Detlefs, could facilitate this.

12 Availability

The sources for CMM are available for anonymous ftp from site `ftp.di.unipi.it` in the directory `/pub/project/posso`. Please address comments, suggestions, bug reports to `cmm@di.unipi.it`.

13 Acknowledgements

Carlo Traverso and John Abbott participated in several meetings and provided valuable feedback on the design. Joachim Hollman provided useful comments on the first implementation. J.C. Faugere provided the idea for this work, adopting a specific memory management in his implementation of the Buchberger algorithm. Discussions with J. Ellis where useful to ensure compatibilty of his proposal with our framework.

References

[Bartlett 88] Joel F. Bartlett "Compacting garbage collection with ambiguous roots" Tech. Rep. 88/2, DEC Western Research Laboratory, Palo Alto, California, February 1988.

[Bartlett 89] Joel F. Bartlett "Mostly-copying collection picks up generations and C++", Tech. Rep. TN-12, DEC Western Research Laboratory, Palo Alto, California, October 1989.

[Boehm 88] H.-J. Boehm and M. Wiser "Garbage collection in an uncooperative environment", Software Practice and Experience, 18(9), 1988, 807-820.

[Breuel 92] Thomas M. Breuel "Personal communication", October 1992.

[Buchberger 85] B. Buchberger, "Gröbner bases: an algorithmic method in polynomial ideal theory", *Recent trends in multidimensional systems theory*, N. K. Bose, ed., D. Reidel Publ. Comp. 1985, 184–232.

[Detlefs 92] D. L. Detlefs, "Concurrent garbage collection for C++", CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, 1990.

[Edelson 92] D.R. Edelson "Precompiling C++ for garbage collection", in *Memory Management*, Y. Bekkers and J. Cohen (Eds.), Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 299-314.

[Edelson 92b] D.R. Edelson "A mark-and-sweep collector for C++", Proc. of ACM Conference on Principle of Programming Languages, 1992.

[Ellis 93] J.R. Ellis and D.L. Detlefs "Safe, efficient garbage collection for C++", Xerox PARC report CSL-93-4, 1993.

[Samples 92] A.D. Samples "GC-cooperative C++", Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 315-329.

[Wentworth 90] E. P. Wentworth "Pitfalls of conservative garbage collection", Software Practice and Experience, 20(7), 1990, 719-727.

[Wilson 92] P.R. Wilson "Uniprocessor garbage collection techniques", in *Memory Management*, Y. Bekkers and J. Cohen (Eds.), Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 1-42.

Safe, Efficient Garbage Collection for C++

John R. Ellis
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
ellis@parc.xerox.com

David L. Detlefs
DEC Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94302
detlefs@src.dec.com

February 24, 1994

Abstract: We propose adding safe, efficient garbage collection to C++, eliminating the possibility of storage-management bugs and making the design of complex, object-oriented systems much easier. This can be accomplished with almost no change to the language itself and only small changes to existing implementations, while retaining compatibility with existing class libraries.

Our proposal is the first to take a holistic, system-level approach, integrating four technologies. The *language interface* specifies how programmers access garbage collection through the language. An optional *safe subset* of the language automatically enforces the safe-use rules of garbage collection and precludes storage bugs. A variety of *collection algorithms* are compatible with the language interface, but some are easier to implement and more compatible with existing C++ and C implementations. Finally, *code-generator safety* ensures that compilers generate correct code for use with collectors.

1. Introduction

We propose adding safe, efficient garbage collection to C++, eliminating the possibility of storage-management bugs and making the design of complex, object-oriented systems much easier. This can be accomplished with almost no change to the language itself and only small changes to existing implementations, while retaining compatibility with existing class libraries.

C++ programmers spend large chunks of their time designing for explicit storage management and tracking down storage bugs, and products are routinely shipped with many such bugs. Dangling pointers, storage leaks, memory smashes, and out-of-bounds array indices are the bane of the C++ programmer and his customers. Some of the most complicated aspects of C++ arise from using constructors and destructors to make up for the lack of garbage collection. Even worse, the need for explicit deallocation of objects cramps the design of modular, reusable interfaces, especially in object-oriented languages like C++.

Despite over a decade of experience using garbage collection for serious application programming in languages like Cedar, Clu, and Modula-2+, many C++ programmers are skeptical of garbage collection's practicality, and C++ vendors are already daunted by the task of efficiently implementing a complicated language that is still evolving. To be successful, garbage collection will have to be introduced slowly and incrementally, in a way that is highly compatible with existing language implementations and class libraries. Programmers, vendors, and the ANSI C++ standards committee will resist any design that requires significant language changes, non-trivial modification of existing class libraries, or global changes to C++ implementations.

Previous attempts at C++ garbage collection have been variously flawed, and they have tended to focus on narrow aspects of the problem. Some have proposed unrealistic language extensions, while others have avoided any modifications of the language or compilers. They all restrict compatibility with existing class libraries and completely ignore safety, assuming that "safe C++" is an oxymoron. But safety is an essential ingredient for eliminating bugs and making garbage collection more usable by the practicing programmer.

Our proposal is the first to take a holistic, system-level approach and the first to provide complete safety. It integrates four technologies:

> a *language interface* specifying how programmers access garbage collection through the language;
>
> an optional *safe subset* of the language that precludes storage bugs and ensures correct use of the collector;
>
> *collection algorithms* already shown to be compatible with existing C++ and C implementations; and
>
> *code-generator safety* that ensures compilers generate correct code for use with garbage collectors.

We recognize the reality of the C++ world. The use of garbage collection is not required—programmers decide which classes should be garbage collected. The one language change is a new type specifier `gc` that enables compatibility with existing libraries; the specifier is easily implemented and has no effect on the type-checking rules. A program can use garbage collection without being written in the safe subset—the programmer decides which parts, if any, of the program should get the automatic protection from bugs. The safe subset consists of 10 compile-time restrictions and 6 run-time checks; these are easily implemented as localized changes to the front ends of current compilers, and the run-time checks can be disabled in production code. The collection algorithms are almost completely compatible with existing implementations, requiring just a small change to code generators to ensure correct operation.

This proposal is deliberately a paper design—it is not yet implemented. There have been decades of experience with garbage collection, including systems programming and application programming languages, and many researchers have built prototype collectors for C++. No more "research" is required—it is now time to consolidate and build consensus on as many of the actual details as possible. A prototype implementation will be necessary to test and polish these details and present a proposal to the ANSI standards committee, but it won't reveal much significant new insight in how to use or implement garbage collection.

A note about language definitions: The ANSI standards committee has not yet finished their standard definition of C++, so we've based our proposal on the current de facto standard, *The Annotated C++ Reference Manual* [Ellis 91], hereafter called the *ARM*. Judging from the partially completed draft standard and the issues the committee has still to resolve, we're confident that our proposal will be compatible with the final language definition.

2. Constraints

To be useful for commercial programming, C++ garbage collection should satisfy the following constraints:

> *Minimal changes:* Too many or too severe changes to the language, its implementations, or programming styles will impede acceptance of garbage collection by the C++ community.
>
> *Coexistence:* Program components using garbage collection must coexist with components not using it.
>
> *Safety:* The rules for correct use of garbage collection should be explicitly defined, and the language and its implementation should provide optional automatic enforcement.
>
> *Portability:* A program using garbage collection should run correctly on all implementations of C++.
>
> *Efficiency:* The more efficient garbage collection is, the more likely it will be accepted.

2.1. Minimal changes

It will be years before garbage collection is widely accepted by the C++ community, and the more changes made to the language or required of its implementations, the longer it will take to get those changes accepted and the less likely C++ garbage collection will succeed. The ANSI standards committee is swamped by hundreds of proposals to "improve" C++, and the simpler a proposed change, the more likely it might be accepted by the committee, by vendors, and by practicing programmers.

C++ vendors are more likely to accept garbage collection if their implementations require at most small changes. They'll resist changes requiring significant changes to the compiler or to the representations of objects and classes.

Programmers are more likely to accept garbage collection if they have to make at most small changes to their programming methodology and style. Programmers tend to be quite conservative and resist change unless they can see immediate, clear benefits.

2.2. Coexistence

Any practical design for C++ garbage collection must allow libraries written without garbage collection or written in other languages such as C or Fortran. A team of programmers wanting to use garbage collection will likely be using libraries written by other teams or companies, and it's unrealistic to expect that all those libraries would be written using both C++ and garbage collection or that the programmers would have access to the libraries' sources.

Making all objects garbage-collected, including objects allocated by existing libraries, is not feasible. First, those libraries may not follow the safe-use rules of the collector, and there is no way to verify safety without access to sources. Even with access, client programmers can't be expected to verify the safety of large libraries. Second, the libraries may not have been compiled by collector-safe code generators, and the chances for problems would lessen if they didn't allocate collected objects (see section 11.2.). Finally, making all objects garbage-collected would change the semantics of destructors, since the collector destroys unreachable objects at unpredictable times. Existing libraries using destructors would break in subtle ways, and C++ programmers and vendors would likely view such a change as too radical.

Experience with systems-programming languages with integrated garbage collection, such as Cedar and Modula-2+, shows that, compatibility issues aside, it is often useful to have two heaps, one for collected objects and one for non-collected objects. The non-collected heap is used for code with ultra-critical performance requirements or, with copying collectors, for objects that can't be relocated for one reason or another.

Thus, a practical design should provide two logical heaps, a collected heap and the traditional C++ non-collected heap. It should be possible to pass collected objects to unmodified libraries written without collection or in another language, and a single library should be able to manipulate both collected and non-collected objects. In particular, objects in the non-collected heap should be able to point at objects in the collected heap, and vice versa. That is, a pointer of type `T*` should be able to reference an object of type `T` allocated in either heap.

For example, suppose a programmer wishes to write a new X Windows application using garbage collection. He'd like to use an X user-interface library written in C by another company that doesn't use garbage collection. The library requires "client data" to be passed and stored in the library's objects. The library itself doesn't interpret the client data, but it holds on to it for the client and passes it back as arguments to call-back functions. The programmer would like to pass collected objects as client data to the unmodified library, without fear that the objects would be prematurely freed.

These constraints preclude extensions to the type system identifying which pointers reference collected objects. Such extensions would require existing libraries to be modified and would effectively prevent libraries from manipulating both collected and non-collected objects.

Note that implementations may represent the collected and non-collected heaps using a single internal heap, with only collected objects being considered for garbage collection.

2.3. Safety

Every garbage collector has a set of safe-use rules that must be followed for its correct operation. For example, a program shouldn't disguise a pointer by xor-ing it with another pointer, because the collector wouldn't be able to identify the pointer's referent.

In standard C++, violating the language's safe-use rules can create hard-to-debug messes. For example, prematurely freeing an object can cause obscure bugs. Garbage collectors dramatically reduce the occurrence of such bugs. But if programmers accidentally violate the safe-use rules, the resulting mess can be even harder to debug than without garbage collection, since collectors can scramble the heap when the rules are violated.

Ideally, the safe-use rules should be enforced automatically by the language, the compiler, and the run-time implementation, with as many rules statically checked as possible. If some of the rules require run-time checks (such as array-subscript checks), those checks should be cheap. However, because storage bugs are so costly to detect and fix, many programmers will gladly pay some amount of run-time overhead for early detection of bugs during development.

However, the automatic static and run-time checking should be **optional**, easily disabled by the programmer at any point during development. Projects with rigorous design and testing methodologies may reasonably decide to trade a marginal bit of safety in production code for increased performance. Also, many old-time C and C++ programmers resist any restrictions on their "freedom" without considering whether such restrictions will improve the final product, and we'd like to encourage such programmers at least to use garbage collection, even if they're not willing to use automatic safety checking.

The compiler must cooperate in following the safe-use rules. In particular, it must ensure that every object reachable from source-level pointers indeed has at least one object-code pointer referencing it, if only in a register or stack temporary. Unfortunately, traditional optimization can generate code in which there is no pointer pointing at or into a reachable object, fooling the garbage collector into prematurely freeing it. Section 11. discusses code-generator safety and how compilers must be modified slightly for garbage collection.

2.4. Portability

The definition of C++ garbage collection should allow programmers to write programs easily that yield the same results on any correct C++ implementation. That is, the garbage-collection safe-use rules should be independent of implementations.

Portability conflicts with efficiency and minimal changes. For example, C++ allows a pointer to be cast to a sufficiently large integer and back again, yielding the same pointer. Totally conservative garbage collectors can handle such casting, since they interpret every word in memory as a potential pointer. But if we want to allow implementations the freedom of using potentially more efficient algorithms (such as partially conservative or copying collectors), then the safe-use rules must encompass those algorithms by restricting the use of certain C++ features.

2.5. Efficiency

Inefficient implementations of garbage collection will impede its acceptance as surely as any set of radical language changes. To be successful, garbage collection needn't be quite as efficient as programmer-written deallocation, since many programmers would gladly sacrifice a little extra run time or memory to eliminate storage bugs quickly and reliably. Though programmers often delude themselves into thinking that they can easily eliminate storage bugs, consider how many programs are shipped with storage bugs and how many months, sometimes years, it takes for those bugs to get fixed.

Recent measurements by Zorn indicate that garbage collectors can often be as fast as programmer-written deallocation, sometimes even faster [Zorn 92, Zorn 93]. Just as many programmers think they can eliminate all storage bugs, they also think they can fine-tune the performance of their memory allocators. But in fact, any project has a finite amount of programming effort, and many, if not most, programs are shipped with imperfectly tuned memory management. This makes garbage collection more competitive in practice.

Efficiency conflicts with minimizing language changes and enabling coexistence. Most previous approaches to garbage collection have relied on non-trivial language support to achieve acceptable performance. In languages such as Cedar, Modula-2+, and Modula-3, programmers must declare which pointers point at collected objects—a collected pointer can't point at a non-collected object, and vice versa. Garbage collectors have relied on such support to scan the heaps more efficiently, to implement generational collection, or to implement reference counting. In general, any language design requiring declaration of pointers to collected objects will often require source modification of existing libraries, which as discussed above makes coexistence of collected and non-collected libraries harder.

Fortunately, recent advances in garbage collection for hostile environments let us strike a practical balance among our goals. Section 8. discusses technology for implementing generational collection and efficient scanning of the heaps without requiring enhanced pointer declarations and sacrificing coexistence.

3. Previous work

No previous proposal or implemented technology meets the constraints outlined above.

3.1. Other languages

Languages like Lisp and Smalltalk have provided safe garbage collection for over two decades. Lisp especially has demonstrated how much safe collection can improve programmer productivity. But other characteristics of those languages have discouraged widespread commercial use.

In the last decade, collection has been successfully integrated into more traditional systems-programming languages like Cedar [Rovner 85a], Modula-2+ [Rovner 85b, DeTreville 90b], and more recently, Modula-3 [Nelson 91]. Unlike C++, these languages were designed with garbage collection in mind, and they refined the notions of garbage-collection safety and providing a safe subset within a larger, unsafe language. But many practicing programmers think the languages are too restrictive, and their implementations prohibit or restrict interoperability with other languages. The C++ subset presented here is noticeably less restrictive (see section 7.).

3.2. Mark-and-sweep collectors

Boehm et al. have implemented a family of conservative mark-and-sweep collectors suitable for use with C++ and C [Boehm 91]. The collectors redefine `new` and `malloc` at link time to allocate from the collected heap. The collectors require no changes to the language and are mostly compatible with current programming styles. The collectors are highly compatible with existing implementations, but they require compiler implementation of code-generator safety, which no compilers currently provide—programmers are on their own to guard against incorrect optimizations, often by disabling optimization entirely. Coexistence is compromised, since all C++ objects are allocated in the collected heap. Safety is not checked—programmers must ensure their programs follow the collector's safe-use rules. Though they are fully conservative, the collectors are surprisingly efficient and quite competitive with explicit, programmer-written deallocation [Zorn 92, Zorn 93]. However, there are as yet no comprehensive measurements of their behavior in long-running programs with large heaps, and unsubstantiated folk wisdom maintains that in practice copying collectors may be more efficient.

Codewright Toolworks has recently started selling a conservative mark-and-sweep collector suitable for C and C++ [Codewright 93].

3.3. Copying collectors

Bartlett et al. have implemented partially conservative copying collectors for C and C++ [Bartlett 89, Detlefs 90, Yip 91]. The collectors require no language changes and are mostly compatible with current programming styles, but programmers must write scanning methods for every class identifying the location of pointers within instances of the class. As with the Boehm mark-and-sweep collectors, programmers must guard against compiler optimizations violating safety. Though the Bartlett collectors provide both collected and uncollected heaps, it isn't possible to pass collected objects to uncollected

libraries directly or store pointers to collected objects in uncollected objects—programmers must write interface stubs that store argument objects in "escape lists" before passing them on to the libraries. But programmers can make mistakes writing these stubs, causing dangling references and storage leaks; often it's very difficult to know when an object can be removed from an escape list. There is no safety checking, and Bartlett collectors require more rules to be followed than Boehm collectors. Programmers mustn't depend on objects having fixed addresses, and they must write correct scanning methods. While writing scanning methods for classes is easy, it's not hard to make a mistake in very large, long-lived, evolving systems maintained by dozens of programmers, and the resulting bugs can be tedious to track down. Finally, though the Bartlett collectors haven't been measured as thoroughly as the Boehm collectors, the measurements that have been made are promising [Detlefs 90, Yip 91].

3.4. Smart pointers

A number of researchers have investigated so-called "smart pointers" as a means of implementing garbage collection purely at the source-language level, without changes to the language or implementations [Edelson 91, Edelson 92, Detlefs 92, Ginter 91]. Using operator overloading and template classes or preprocessors, the collectors get notified whenever a smart pointer is created, destroyed, or assigned. But smart pointers don't entirely mimic the functionality of standard pointers. Given class `T` derived from `S`, a smart pointer to `T` can't be assigned to a smart pointer to `S`. Such widening casts are an essential feature of C++, and prohibiting them wouldn't be practical. Using a preprocessor to work around this limitation effectively changes both the language and its compilers. Since pointers to collected objects must be explicitly declared as smart pointers, coexistence with existing libraries is precluded. Smart pointers provide no automatic safety checking, and they can't prevent unsafe code-generator optimizations without doubling the size of pointers, adding run-time overhead, and relying on the implementation-dependent semantics of `volatile` [Detlefs 92]. Ginter proposes some language changes that would make smart pointers feasible, but that defeats the original goal of avoiding language changes [Ginter 91]. In sum, smart pointers are actually rather dumb.

3.5. Pointer declarations

Samples has recently proposed adding two new type qualifiers to C++ that declare in which heap an object should be allocated and that identify which pointers may point at collected objects and which may point into the middle of objects [Samples 92]. The proposal involves non-trivial changes to the language's type-checking rules. Though the proposal would be compatible with conservative and partially conservative collectors, realizing the efficiency gains enabled by the declarations would require changing the object representations used by current compilers. As discussed in section 2.2., requiring declaration of pointers to collected objects inhibits coexistence with existing libraries, since the libraries would need source changes to coexist with collected objects. Though the static type checking rules help enforce safety, there is no complete safety checking—Samples believed that wasn't feasible with C++.

Samples's proposal is designed to allow a wide range of collection algorithms, including non-conservative algorithms. With sufficient compiler support and changes to the representation of objects, the declarations can help the collector identify which pointers may point at collected objects and of those, which may address the interiors of objects. This support may reduce the collector's cost of following pointers during a collection (see section 10.).

As yet there are no detailed measurements indicating how much efficiency pointer declarations would buy, though Zorn's measurements of the totally conservative Boehm collector suggest that even without declarations, collectors can compete with traditional explicit deallocation [Zorn 92, Zorn 93]. Presumably, a version of the Boehm collector using precise scanning of heap objects (via type maps) would be even more efficient. Thus, pointer declarations most likely aren't required to provide acceptably efficient collectors.

In summary, Samples's proposal may allow for somewhat more efficient garbage collectors but at the cost of non-trivial language and compiler changes, and of sacrificing coexistence.

3.6. Development tools

Tools such as CenterLine [CenterLine 92] and Purify [Pure 92] detect storage bugs during development. CenterLine provides an interpreter that can catch almost all such bugs, while Purify uses link-time code modification to catch most heap-storage bugs (but not stack- or static-storage bugs). Since the tools slow down programs considerably when providing full error detection (CenterLine by a factor of 50, Purify by a factor of two to four) and use noticeably more heap memory, they are most appropriate for testing programs where execution speed is not too important. The tools are too slow for many kinds of CPU-intensive testing or use in production releases.

While such tools are very useful, programmers must still spend considerable time designing, implementing, and debugging explicit memory deallocation. Safe garbage collection greatly reduces that design and debugging time, and it can be used throughout development and release with little or no sacrifice in performance. More importantly, garbage collection simplifies the interfaces of complicated systems and enhances reusability.

4. Language interface to garbage collection

The language interface specifies how programmers access garbage collection through the C++ language. The complete specification of the language interface appears first, followed by a design discussion.

4.1. Specification

Objects may be allocated in one of two logical heaps, the collected heap and the non-collected heap. *Collected objects*, objects allocated in the collected heap, will be automatically garbage collected when they are no longer accessible by the program; *non-collected objects*, objects allocated in the non-collected heap, must be explicitly deallocated using `delete`. Objects in one heap may contain pointers to objects in the other heap.

A new kind of type specifier, a *heap specifier*, tells `new` in which heap it should allocate. The `gc` specifier selects the collected heap, and `nogc` (the default) selects the non-collected heap. For example:

```
gc class A {...};
typedef gc char B[10];
nogc class C {...};
typedef int D[5];
```

The expressions `new A` and `new B` allocate in the collected heap, and the expressions `new C` and `new D` allocate in the non-collected heap.

Like a storage-class specifier, a heap specifier applies to the object or name being declared. For example, the declaration `gc char a[10]` declares `a` to be a garbage-collected array of characters, not an array of garbage-collected characters.

The types `T` and `gc T` are two different types, but an expression of type `gc T` can be used wherever an expression of type `T` is allowed, and vice versa; thus, an expression of type "pointer to `gc T`" can be used wherever an expression of type "pointer to `T`" is allowed, and vice versa. In particular, a pointer of type `T*` may point at an object of type `gc T`, and an object of type `gc T` can be declared static or automatic (in which case `gc` is ignored).

The heap specifier is included in the type-safe linkage of a name, ensuring that all occurrences of it have the same meaning.

A declaration of a non-class type without a heap specifier defaults to `nogc`. Similarly, a declaration of a class with no base classes and no heap specifier defaults to `nogc`. But a declaration of a derived class with no heap specifier inherits the heap specifier of its base classes, and it is an error if the heap specifications of the base classes conflict. An explicit heap specifier always overrides the specifiers of the base classes (even if they conflict). Examples:

```
gc class A {};
class B {};
gc class C: B {};       /* ok */
class D: A, B {};       /* error */
gc class E: A, B {};    /* ok */
```

A `gc` class may not overload `operator new` or `operator delete`; an inherited `operator new` or `delete` is ignored.

The expressions `new T` and `new T[e]` allocate in the heap selected by `T`'s heap specifier.

Regardless of which heap is selected, `new T` and `new T[e]` invoke `T`'s constructors in the standard way.

When the garbage collector discovers that a collected object is inaccessible to the program, it will invoke the object's destructor before recycling its storage. This allows programmers to define clean-up actions that release the resources of unused objects. The destructor will be called asynchronously with respect to execution of the main program. See section 5. and appendix B for the precise semantics of clean-up and a design rationale.

If `e` points to a collected object, the statement `delete e` invokes the object's destructor immediately, returning after it finishes, and the collector won't invoke the destructor later when it collects the object. Deleting a collected object is a hint to the implementation that it may reuse the object's storage, but implementations can ignore the hint. As with non-collected objects, it is illegal to reference a deleted collected object, though implementations aren't required to check for that.

4.2. Rationale

The design of the language interface provides coexistence of collected and non-collected libraries using the smallest possible change to the language. As discussed in section 2.2., coexistence with existing libraries requires two logical heaps and some way for the programmer to select between them, and adding heap specifiers to the language is about the simplest way to do that. Heap specifiers don't affect the language's type-checking rules, so collected and non-collected objects can be freely intermixed, and collected objects can be passed to existing non-collected libraries.

Heap specifiers provide programmers with some ability to adapt old code to use the collector. Given a non-collected class `C`, an instance of `C` can be allocated in the collected heap using the expression `new gc C` or by defining a type name:

```
typedef gc C CGC;
```

and using the expression `new CGC`. Thus, if a program must import a library that doesn't use garbage collection, the program can still create collected instances of the library's classes.

The program can also derive a collected class from a library's non-collected class `C`:

```
gc class D: C {...};
```

Note that if class `C` has a destructor, the situation is more complicated, since `C`'s destructor may unexpectedly be invoked asynchronously by the collector (see section 5.2.).

Unlike traditional garbage-collected languages, our proposal allows `delete` to be applied to collected objects. We believe that almost all C++ code written from scratch will have no need to use `delete`; indeed, the safe subset prohibits its use, since it is inherently unsafe. But there are two important reasons for allowing `delete` of collected objects.

First, programmers adapting old code may want to use garbage collection as a backup to catch existing storage leaks, while making as few changes as possible. This suggests that `delete` applied to a collected object should, at a minimum, invoke the object's destructors immediately.

Second, programmers writing new garbage-collected code may want to use `delete` as a performance hint for the collector, suggesting that some particular objects can be deleted immediately. Depending on the implementation, this may significantly reduce the load on the collector. Implementing immediate deletion is easy with mark-and-sweep algorithms, but we don't know how copying algorithms might take advantage of the deletion hints.

Allowing explicit deletion of collected objects lets programmers optimize resource-critical sections of their systems. Often, large systems have small, circumscribed sections that are responsible for large fractions of total storage allocated, and while it may be hard to identify *all* objects that can be safely deleted, it's often easy to identify many or most objects which can. Programmers can use explicit deletion to make the easy safe optimizations, relying on garbage collection to catch any leftover objects that were missed or hard to delete safely.

In both these scenarios, `delete` is inherently unsafe and its use requires care to avoid bugs that can't be detected at compile time or run-time. The program may prematurely delete an object, creating dangling pointers. Also, when adapting old code, the programmer must realize that the collector will invoke the destructors of collected objects asynchronously and that the old code may not be prepared for that.

Despite these problems, we think that allowing deletion of collected objects will be sufficiently useful that it shouldn't be outlawed. It's very easy to implement (collectors can simply ignore the deletion hint). It also follows the spirit of C++, providing programmers with a dangerous power tool; those programmers who don't want to cut their hands off can use our safe subset, which prohibits the use of `delete`.

Finally, declaring a class to be `gc` in effect supplies the class with the garbage collector's allocation and deallocation methods. Thus, there's no reason for a `gc` class also to have an overloaded `new` or `delete`, and any attempt to do so must be a programming mistake.

4.3. Alternatives to the gc keyword

Some critics have suggested using the placement syntax of operator `new` to control whether objects are collected or not. This would require no language changes, but it would push onto clients the responsibility for deciding where objects should get allocated by default. Given a collectible class `T`, either:

> clients of `T` must always remember to specify the `gc` placement when they write `new T`; or
>
> `T` must provide static member functions for creating new single instances and new array instances.

With the first option, clients can easily forget to specify placement, and it is also more verbose, for example, `new (gc) T`.

The second option is counter to the design of C++, in which clients are expected to write `new T` and `new T[e]` to create instances of `T` and the language provides the implementor of `T` with mechanisms for properly initializing instances when `new` is invoked. The option would give collectible classes a different look and feel from non-collectible classes, and that might impede the acceptance of garbage collection.

Further, whether an object is collectible affects the semantics of its destructors, since collector-invoked destructors run asynchronously. An implementor of a class with a destructor would like some way to communicate to clients whether instances of the class can be safely collectible. In our proposal, the heap specifier allows the class implementor to provide clients with the correct default placement. If the language provides no heap specifier, then clients are on their own for using the correct placement.

To hide the deficiencies of placement syntax, others have suggested using a base class `gc` to specify which objects should be collected:

```
class gc {public:
    void *operator new(size_t sz);
    operator delete(void*);
    virtual ~gc();};
```

Objects derived from `gc` would be collected. The class has a virtual destructor that registers itself with the collector—when a `gc` object is collected or deleted, the destructors of all the derived classes will be called.

Compared to adding a `gc` keyword, using a base class requires no changes to the language. However, the base-class approach has a number of disadvantages that, in balance, yield an inferior design.

First, programmers must still use the error-prone placement syntax to allocate collected instances of non-class types.

Second, the virtual destructor `~gc()` adds unnecessary overhead. It imposes the storage overhead of virtual functions on all collected objects, even if they don't have any other virtual functions; this violates

an important design principle of C++. Also, `~gc()` registers itself as a clean-up function with the collector, even if derived classes don't actually provide destructors. Registering a clean-up function imposes non-trivial storage and time overheads (see section 5.1. and appendix B).

To avoid the overhead, we'd need two base classes, one for normal collected objects and one for collected objects requiring clean-up:

```
class gc {public:
    void *operator new(size_t sz);
    operator delete(void*);};

class gccleanup: public gc {public:
    virtual ~gc();}
```

But this adds more complexity for programmers and can lead to more mistakes. A programmer might add a destructor to a collected object and forget to change its base class to `gccleanup`.

Finally, programmers will often be forced to use multiple inheritance to make objects collected. For example, to derive a collected class from an existing non-collected class would require multiply inheriting the base class `gc`. Many existing class libraries provide their own root base class, and making collected versions of their classes would require the multiple base `gc`.

Many implementations of multiple inheritance impose non-trivial allocation overhead on small objects, which would discourage the creation of small collected objects. Further, many C++ programmers object to multiple inheritance on principle. Justified or not, their resistance could impede acceptance of garbage collection.

Considering all this, we think adding a `gc` keyword is the best design. It's trivial to implement compared to the other costs of implementing garbage collection, and certainly the ANSI C++ committee has not shied away from adding new keywords to support important new features. However, if the `gc` keyword is rejected in the end, the base-class approach would be better than not having garbage collection at all.

5. Object clean-up and weak pointers

Two closely related facilities, *object clean-up* and *weak pointers*, allow programs to track when unreachable objects are freed by the garbage collector. Object clean-up lets the programmer specify actions to be taken when an object is no longer accessible and about to be garbage collected. Weak pointers enable the construction of caches of objects that don't require clients to indicate when they are finished using cached objects.

5.1. Object clean-up

When an object is no longer used by the program and is about to be freed, it is often necessary to clean up the object by releasing resources it holds or removing it from a global data structure. For example, if an object contains an open file handle, the object's clean-up might close the file. Or if an object contains a window handle, the clean-up might release the handle back to the window system. In general, if an object contains some resource controlled by another program, the operating system, or a non-collected library, a clean-up action can release the resource when the object is garbage collected.

C++ supports clean-up with destructors—when an object of class `T` is about to be freed, the destructors of `T` and its base classes are applied to the object. Automatic objects are freed when their scope is exited; static objects are freed when the program terminates; and heap-allocated objects are freed when `delete` is called on the object. (Note that storage allocated by an overloaded `new` should be released by an overloaded `delete`, not by a destructor.)

In our proposal, garbage-collected objects are also cleaned up using destructors. If the programmer wants objects of `gc` class `T` to be cleaned up when they are about to be garbage collected, he writes a destructor `~T()` that performs the clean-up. When the collector determines that an object is unreachable by the program, it calls the object's destructor before freeing it. The collector considers an object

unreachable if it can't be accessed by following a path of pointers starting from static variables or automatic variables of active functions.

If collected object `B` is reachable from collected object `A`, then `A`'s destructor will be invoked before `B`'s destructor. This ensures that `A`'s destructors will see a fully formed, non-cleaned up `B`. `B` will be cleaned up only after `A` has been collected. If `A` and `B` have non-empty destructors and each is reachable from the other (that is, they form a cycle), neither will be cleaned up or collected.

An explicit clean-up function `f` can be registered for an object `t` of type `T` using the standard interface `CleanUp`:

```
CleanUp<T, void>::Set(t, f)
```

(The default clean-up function for an object with a non-empty destructor simply calls the destructor.) Clean-up for a particular object can be disabled entirely by setting the clean-up function to null:

```
CleanUp<T, void>::Set(t, 0)
```

Programmers can force an object's clean-up to be invoked immediately either by calling

```
CleanUp<T, void>::Call(t)
```

or `delete t` (whose implementation calls `CleanUp::Call`). An object's clean-up is called at most once, unless it is explicitly re-enabled by calling `CleanUp::Set`.

There is no guarantee that the collector will detect every unreachable object and invoke its destructors. Conservative collection algorithms find almost all unreachable objects, but not all of them, and any algorithm likely to be used for C++ in the next several years will almost certainly use a conservative scan of stacks and registers. Thus, programmers should treat object clean-up as a mechanism for improving resource usage, and they should not rely on having clean-ups applied to 100% of their objects.

Finally, if programmers need some action to occur on heap objects when the program exits, they should use a termination service like that described by Stroustrup [91, page 466]. Termination actions are not the same as destructors—the destructor of a heap-allocated object won't necessarily be called when the program exits.

The precise semantics of object clean-up and the `CleanUp` interface are presented in appendix B.

5.2. Clean-up asynchrony

Because the collector may run at arbitrary times, a collected object's destructor may be invoked asynchronously with respect to the main thread of execution. This isn't a problem if the destructor side-effects data that is reachable only from the object, since by definition when the destructor is invoked, no other parts of the program can access the object. But sometimes a destructor must access global data or other objects that are still accessible to the rest of the program, and in these cases such access must be synchronized to avoid races.

In multi-threaded environments, synchronizing concurrent access is straightforward using well-understood techniques. In general, programmers must synchronize access to all global data, so synchronizing destructors takes little extra effort.

But in a traditional single-threaded environment, programmers usually assume there is no concurrency, and a naively programmed destructor could access inconsistent data. In these environments, destructors can synchronize using queues provided by the standard interface `CleanUp`. You can declare a clean-up queue for instances of a type `T`:

```
CleanUp<T, void>::Queue q;
```

Calling `q.Set(t)` tells the collector that when `t` becomes unreachable it should enqueue it on `q` instead of calling its destructor. The program can poll `q` periodically at safe points by calling `q.Call()`. Each such call removes the first object from the queue and calls the object's destructor; `q.Call()` does nothing if `q` is empty. An example of this technique is presented in section 5.4..

5.3. Clean-up rationale

Advocates and critics of C++ garbage collection have been gnashing teeth over object clean-up and destructors. Three issues arise: should there be object clean-up at all; if so, how should clean-ups be specified syntactically; and in what order should clean-up functions be applied?

Should the collector provide object clean-up? There's been over a decade of experience using garbage-collected languages such as Lisp, Cedar, Smalltalk, Clu, and Modula-2+ to build long-lived applications, servers, and operating systems. In these environments, programmers have found object clean-up indispensable for managing in-memory caches of objects and releasing resources provided by other programs, servers, operating systems, and non-collected libraries [Hayes 92]. There's every reason to expect that object clean-up would be equally useful in systems built with C++.

In the second edition of *The C++ Programming Language* [Stroustrup 91, page 466], Stroustrup provides widely quoted arguments against collector-based clean-up. First, he argues that garbage collection simulates an infinite memory from which objects never get deleted; since the objects are never deleted from the (simulated) infinite memory, the collector shouldn't invoke their destructors. This argues by analogy without considering whether the analogy is valid, and like most such arguments it fails to address the basic question: What is most useful for building systems? In fact, most programmers using collector-based languages view the collector as automating calls to `delete`, and under this analogy, it's quite sensible for those calls to invoke clean-ups.

Stroustrup suggests that destructors (clean-ups) should only be invoked as the result of explicit calls to `delete`. But in general this requires programmers to determine when an object is no longer being used before invoking their clean-ups. Such a requirement defeats the major purpose of garbage collection, removing that burden from programmers. (Note that our design allows the programmer to force immediate invocation of destructors by calling `delete`.)

Stroustrup then implies that program-termination actions can replace most collector-driven clean-up actions. (Termination actions are registered functions that get applied to an object when the program terminates.) But of course, termination actions are different from timely clean-up—they have different purposes, and programmers want both. Clean-up actions release resources in a timely manner during program execution, whereas termination actions ensure some action is taken *only* when the program exits. (Note that programs such as servers never exit.)

Many critics are discomfited by asynchronous clean-ups. Stroustrup argues that asynchronous collector-driven clean-up is "hard to program correctly and less useful than is sometimes imagined". But we show in sections 5.2. and 5.4. how to program asynchronous clean-ups correctly with no fuss or muss, even in traditional single-threaded C++ environments. And programmers in those other collector-based languages would certainly dispute the implication that clean-up isn't very useful.

Note that destructors may be applied at unexpected times even in standard C++. In the *ARM*, the exact point at which destructors of temporary objects are called is implementation-dependent. The ANSI standards committee is currently wrestling with the problem of when destructors of automatic objects should be invoked, and no matter what they decide, this issue will continue to hold subtle surprises for the unsuspecting programmer.

Interestingly, most examples illustrating problems with C++ destructors involve freeing storage. The most common use of destructors is to free storage, and garbage collection eliminates the need for such destructors. Based on experience with other collected languages, very few collected classes will need explicit clean-up actions. Thus in practice, introducing garbage collection will eliminate most uses of destructors and simplify their use.

How should the syntax of clean-ups be specified? There are two choices for specifying clean-ups: functions explicitly registered with the collector, or C++ destructors. We think destructors are somewhat better, though explicitly registered functions would be adequate.

In languages such as Cedar and Modula-3, programmers write clean-up functions and register them with the collector. Since this requires no syntactic support from the language, some critics suggest this is the best way of specifying clean-ups.

But registered clean-up functions require non-trivial programmer conventions to support modularity and class derivation. Consider a class `U` derived from `T`, with both classes desiring to clean up their

private members. Any convention must allow U and T to register clean-up functions independently in their respective constructors, while ensuring that U's clean-up is called before T's. Further, each class's clean-up function must also remember to call its members destructors.

In light of this, perhaps the best convention would put the clean-up actions for a class in its destructor and register a clean-up function that simply calls the destructor:

```
class T: S {
    T() {
        CleanUp<T, void>::Set(this, CallDestructor);
        ...};
    static void CallDestructor(void* d, T* t) {
        t->T::~T();};
    ~T() {  /* clean-up actions for T */
        ...}
    ....
    };
```

As long as the base classes of T also follow this convention, their clean-up actions will be invoked in the proper order, after T's clean-up actions. Even if the base classes also register clean-ups, all objects of class T will end up with the clean-up function `T::CallDestructor`, since T's constructor runs after its base-class constructors. `T::CallDestructor` calls `T::~T()`, which calls the destructors of the base classes after executing the body of `~T()`.

Compared to our design that registers destructors automatically, this convention has several minor disadvantages. First, programmers can make clerical mistakes, forgetting to register a clean-up function in every constructor of class. Such mistakes may be more likely in classes with many constructors or when new constructors are added to an old class by a programmer who isn't the original author. Second, `CleanUp::Set` may be called several times during an object's construction, whereas one call is sufficient. In a typical implementation, the runtime cost of each extra call could be non-trivial, roughly the same as the cost of the allocation itself. Third, the construction of static, automatic, or non-collected heap instances of such classes will call `CleanUp::Set` unnecessarily (`CleanUp::Set` would do nothing if passed non-collected objects). Finally, the convention is more verbose, and programmers would get annoyed that the language doesn't register the destructors automatically. Destructors get called automatically for static, automatic, and non-collected heap objects, so why exclude collected objects?

Some critics (reportedly including Stroustrup) say that having the garbage collector invoke destructors asynchronously changes the semantics of the language, and thus existing applications would break. This argument assumes a different model for adding garbage collection to the language: `new` would be redefined to allocate *all* objects from the collected heap. Under this design, existing code would indeed sometimes break, since such code often depends on having destructors invoked synchronously at somewhat well-defined points.

However, our design is different: We've argued in section 2.2. that, for several reasons, coexistence with existing libraries requires both collected and non-collected heaps, and that by default `new` continues to allocate from the non-collected heap. Thus, the semantics of destructors for non-collected objects is *not* changed, and existing code continues to execute correctly side-by-side with new code written to use garbage collection. We've optimized our design for writing new code to use garbage collection, while retaining strict compatibility with old code.

Note that if a collected class B is derived from a non-collected class A, A's destructor could get invoked asynchronously when instances of B are collected, and the destructor may not be prepared for that. In this case, the behavior of non-collected objects hasn't been affected, though the behavior of the collected instances of B is less than ideal. We weren't willing to flatly prohibit deriving a collected class from a non-collected class with a destructor, since we thought that in many cases that could be useful, and that this situation would arise only when trying to allocate collected instances of a non-collected class exported by an existing library. The compiler can of course give a warning, and the programmer can use clean-up queues to control the invocation points of the destructors or he can override the destructor with his own clean-up function.

What order should clean-up functions be applied? With garbage collection, the programmer has little control over the precise order in which objects are deleted and their destructors invoked. Instead, the language interface provides a strong ordering guarantee: The collector invokes the destructors of collected objects in topological order. That is, if object `B` is reachable from object `A`, then `A`'s destructor will be invoked before `B`'s. This ensures that when `A`'s destructor executes, all the objects referenced by `A` will have defined, undestroyed values.

However, cycles of collected objects with clean-up functions are problematic. If `A` and `B` are reachable from each other, then destroying either one first will violate the ordering guarantee, leaving a dangling pointer. If the collector breaks the cycle arbitrarily, programmers would have no real ordering guarantee, and subtle, time-dependent bugs could result. To date, no one has devised a safe, general solution to this problem [Hayes 92].

In our design, cycles of objects with clean-up functions will never be collected or cleaned-up. This approach, based on the Boehm collector and Modula-3, at least maintains safety. Implementations are encouraged to have debugging modes that warn programmers when cycles are detected.

Luckily, experience with previous systems shows that cycles of objects with clean-up functions are rare. Most objects don't require clean-up at all, and of those few that do, cycles are very infrequent.

We expect that experience to carry over to C++. Though today many C++ objects have non-empty destructors, most of those destructors exist solely to manage storage. Thus, most collected objects will have empty destructors and the restrictions on cycles won't apply to them.

5.4. Weak pointers

Weak pointers allow a class to track which objects are being used by other parts of the program. The collector ignores weak pointers when tracing reachable objects, so a weak pointer to an object won't prevent the object from getting collected. The most common use of weak pointers is to build caches of objects in which cached objects are automatically deleted by the collector when clients no longer reference them. In contrast to the traditional way of implementing such caches, clients of weak-pointer caches need not tell the caches when they are finished using an object.

For example, suppose a window server contains a cache of in-memory fonts, keyed by font name. Fonts consume a lot of memory, so when clients of the window server no longer reference a font, it should be deleted automatically from the cache, without requiring notification from clients.

Weak pointers are defined by the `WeakPointer` template class (appendix B); no special language support is needed. A weak pointer is constructed from a normal pointer `t` of type `T` using the constructor:

```
WeakPointer<T> wp(t);
```

The `Pointer` method translates a weak pointer back to a normal pointer:

```
T* t1 = wp.Pointer();
```

`Pointer` returns the original pointer, unless the weak pointer `wp` has been *deactivated*, in which case it returns null. The collector deactivates a weak pointer when it garbage-collects the referenced object; that is, when the object becomes unreachable by paths of normal pointers from static variables and automatic variables of active functions.

In our example of a window server's font cache, the server could implement the cache as a table of pairs <font name, weak pointer to font>. Because the table uses only weak pointers to reference its fonts, it won't cause those fonts to be retained in memory by the collector—a font will remain uncollected only if a client still references it.

Here's a sketch of the `FontCache`:

```
class Font;

class FontCache {
    Table<char*, WeakPointer<Font> > table;
public:
    Font* Get(char* fontName);}

FontCache fontCache;
```

The `Get` method returns a font named `fontName`, reading it into memory from disk if it isn't in the cache. Its implementation looks like:

```
FontCache::Get(char* fontName) {
    WeakPointer<Font> wp;
    if (table.Get(fontName, wp)) {
        Font* font = wp.Pointer();
        if (font != 0) return font;}
    Font* font = Font::ReadFromDisk(fontName);
    table.Put(fontName, WeakPointer<Font>(font));
    return font;}
```

`Get` looks in the table for an entry keyed by `fontName`. If there is such an entry, and the font referenced by the entry's weak pointer hasn't yet been garbage collected, `wp.Pointer()` will return a non-null `Font*`, which is returned to the client. If `wp.Pointer()` returns null, that means that clients no longer reference the corresponding font and it has been garbage collected. In this case, and in the case of no entry at all for `fontName`, `Get` reads the font from disk and installs it in the table before returning it.

Note that, over time, the table could fill up with entries whose fonts have been garbage collected. Often, programmers can ignore this problem, since the table entries themselves are small and there often aren't that many entries. If it is a problem, though, there are a couple of straightforward solutions.

First, `Get` could scan the table whenever it fills up, deleting entries whose weak pointers have been deactivated (`wp.Pointer() == 0`). Assuming the table is a dynamically growing hash table, this would increase the time cost of the hash table by a small constant factor.

Alternatively, the table can use object clean-up of fonts. A font's destructor can delete the corresponding entry from the cache:

```
Font::~Font() {fontCache.Delete(this);}
```

However, the collector may call the destructor during the execution of some other operation on `fontCache`, creating a race condition. So we must use a clean-up queue (section 5.2.) for synchronization:

```
class FontCache {
    CleanUp<Font, void>::Queue q;
    ...};
```

When `Get` adds a font to the cache, it calls `q.Set(font)`, telling the collector that when `font` becomes unreachable by normal pointers, it should enqueue it on `q` rather than invoking its destructor. In addition, `Get` calls `q.Call()` before doing its cache look-up, invoking the destructors of any inaccessible fonts enqueued on `q`, thereby removing them from the table. The enhanced `Get` looks like:

```
FontCache::Get(char* fontName) {
    while (q.Call());    /* call ~Font() in a safe place */
    WeakPointer<Font> wp;
    if (table.Get(fontName, wp)) {
        Font* font = wp.Pointer();
        if (font != 0) return font;}
    Font* font = Font::ReadFromDisk(fontName);
    table.Put(fontName, WeakPointer<Font>(font));
    q.Set(font);         /* Set font's clean-up queue */
    return font;}
FontCache fontCache;
```

The precise semantics of weak pointers and their interaction with object clean-up is specified in appendix B.

6. Safety

Any proposal for C++ garbage collection must define the language rules programs must obey to ensure the correct use of garbage collection. A program following these rules is *GC-safe*. Programmers and C++ implementors need a precise language definition of GC-safety to ensure that programs can run on any conforming C++ implementation (subject to memory availability). That is, the GC-safe rules provide a standard interface between the program and the garbage collector, allowing many different programs to run with many different collector implementations.

Our definition of GC-safety is broad enough to encompass all the major families of collector algorithms, yet simple enough for working programmers. To write a portable GC-safe program, the programmer need only follow the usual C++ portability rules plus one additional restriction.

6.1. Definition of GC-Safety

We've adopted Owicki's approach to defining GC-safety [Owicki 81]. The specification of GC-safety is a promise to the program: if all the program's actions are "legitimate", then the garbage collector will remain "invisible" to the program. A garbage collector is invisible if it doesn't free any objects still in use by the program and it doesn't make invalid changes to those objects. The rest of this section defines "legitimate" program actions.

Legitimate program actions must maintain three general properties: separation of memory, visibility of collected pointers, and visibility of assignments. (In what follows, a "collected pointer" is a pointer to a collected object. Also, by "pointer" we mean both C++ pointers and C++ references.)

Separation of memory. The program shouldn't change memory locations belonging to the collector. If it did, say by overwriting structures used to maintain the collected heap, the collector might accidentally reuse the storage of an object still in use, or it might relocate just part of an object.

Further, the program should create new collected objects or new non-collected objects containing collected pointers only by declarations or by invoking `new`. If the program acquired new memory through some mechanism unknown to the collector and then stored collected pointers in that memory, the collector might not be able to find those pointers, and it might prematurely free an object.

Visibility of collected pointers. All collected pointers should be visible at all times. That is, they should be stored in variables, members, or array elements declared with type "pointer", or they should be the results of expressions whose type is "pointer". The locations of pointers within an object are fixed by its declared type (the declaration or the argument to `new`).

A garbage collector needs to know exactly where all collected pointers are located at all times, so that it can determine which objects are still in use and possibly relocate those objects. If a pointer is hidden, say, by casting it to an integer or storing it in a location not declared to be a pointer, then the collector may be fooled into thinking its referent object is no longer in use and freeing it prematurely. Even if some other visible pointer also points at the object, a relocating collector couldn't correctly update the hidden pointer after moving the object.

Visibility of assignments. All assignments of collected pointers must be visible to the garbage collector. That is, if a variable, member, or array element currently contains a collected pointer, or if its value is to be changed to a collected pointer, then it must be changed either by initialization to a pointer value or by an assignment expression in which both the lvalue and rvalue have type "pointer". Some incremental collector algorithms rely on the compiler to generate special code for pointer assignments. Changing collected pointers through some mechanism other than a pointer-typed initialization or assignment (for example, `memcpy`) would hide the assignment from the collector, perhaps causing it to prematurely free the object referenced by the rvalue.

For the purposes of determining safety, we assume that all overloaded assignment and `new` operators have been expanded to their definitions and that all assignments of whole objects have been expanded into their equivalent member-wise assignments.

6.2. Writing portable GC-safe programs

The purpose of GC-safety is to provide a set of rules that let programmers write portable programs that run correctly on many different collector implementations. Despite the apparent complexity of the definition of GC-safety, it's straightforward to write a portable GC-safe C++ program. The programmer need only follow the usual C++ portability rules plus one additional restriction.

A program is guaranteed to be GC-safe if it follows these rules:

> It doesn't execute any of the constructs listed below that the *ARM* labels "undefined" or "implementation-dependent".
>
> It doesn't cast an integer to a pointer, unless the integer resulted from casting a non-collected pointer and the referent of the pointer is still allocated at the time the integer is cast back to a pointer.

(A program that doesn't follow these rules may still be GC-safe on particular implementations.)

The following undefined or implementation-dependent constructs could, perhaps in combination, violate GC-safety on some implementations. The constructs are labeled with the corresponding section of the *ARM*:

> accessing an uninitialized variable, member, or array element (8.4)
> accessing a union member after a value has been stored in a different member of the union (5.2.4)
> dereferencing a null pointer
> accessing a dangling pointer or reference (5.3.4)
> applying `delete` to a pointer not obtained from `new` (5.3.4)
> illegal pointer arithmetic—adding to a pointer not referencing an array element, or arithmetic resulting in a pointer outside the bounds of the array (except for one past the last element) (5.7)
> a subscript expression whose equivalent in pointer arithmetic is undefined (5.2.1)
> all casts to types containing pointers, references, and functions, except legal widening casts ("up-casts"), legal narrowing casts ("down-casts"), casts between pointer types and `void*`, and the casts from integers to pointers described above (5.4)
> casting a pointer to an integer (5.4)
> exiting a value-returning function without an explicit `return` or `throw` (6.6.3)
> using variadic functions (ellipsis) incorrectly (8.3)

By definition, it's impossible to know the behavior of undefined or implementation-dependent constructs without reference to a particular implementation. Thus, a programmer writing a truly portable program must avoid such constructs regardless of whether he's using garbage collection.

Obviously, some of the constructs have well-defined behavior on some implementations. To decide whether they are GC-safe on a particular implementation, a programmer would have to refer to the

general definition of GC-safety and any "specifications" provided by the implementation's vendor. For example, with a fully conservative mark-and-sweep collector, a program could safely hide collected pointers by casting them to integers and it could copy pointers using `memcpy`.

Casting an integer to a pointer could, in general, violate pointer visibility by hiding the pointer as an integer. If the resulting pointer is invalid, dereferencing it could overwrite memory locations belonging to the collector, thus violating separation of memory.

Casting a pointer to an integer yields an implementation-dependent result (*ARM* section 5.4), but programmers often assume that repeated casting of the same pointer will yield the same integer. Of course, this is no longer true with a relocating collector, so programmers wishing to write portable programs should avoid depending on the results of such casts. Though technically, these casts don't violate GC-safety, they violate the spirit of "invisibility" of garbage collection.

6.3. Pointer validity

Our definition of GC-safety allows pointer variables to contain invalid values such as dangling pointers created by `delete` or pointers fabricated by illegal casting. This definition requires collectors to check the validity of every pointer discovered in a variable or object as the collector traces out all live objects. Also, invalid pointers may cause excess storage to be retained by the collector if they happen to point at storage reused for collected objects.

Some have suggested a stronger requirement, that all pointer-valued variables, members, elements, and expressions should evaluate to valid pointers to allocated objects. Most previous garbage-collected languages have required this stronger pointer validity. Pointer validity has some appeal, since if collectors could assume every pointer followed is valid, they might avoid some validity checks and the structures needed to support them, and invalid pointers couldn't accidentally retain excess storage.

There are several counter-arguments to the stronger requirement of pointer validity. First, pointer validity won't in fact save the overhead of validity checks. As we argue in section 2.2., a practical language proposal must allow for arbitrary interior pointers with no special type declarations, and it cannot use type declarations to distinguish between collected and non-collected pointer values. The data structures needed to handle interior pointers and to distinguish between collected and non-collected objects suffice for checking pointer validity, and the validity checks will cost at most an extra instruction per pointer followed (compared to the ten or more instructions needed to map interior pointers to base pointers—see section 10.1.).

Second, pointer validity won't cause less excess storage to be retained. Regardless of whether pointer validity is required, programmers must null out all pointers to unused collected objects to ensure the objects will be freed. If the programmer forgets to null out pointers, excess storage will be retained.

Third, pointer validity would prohibit coexistence with a fair amount of existing C++ code that leaves large numbers of invalid pointers lying around, unused. For example, some libraries use overloaded `new` and `delete` to deallocate a huge collection of objects simultaneously, thus creating large numbers of dangling pointers that are never subsequently dereferenced. As another example, low-level code often creates pointers that don't appear to reference allocated objects.

Fourth, pointer validity requires the argument to `delete` be the last remaining pointer to the object being deleted. Maintaining this property would complicate destructors for non-collected circularly linked structures, requiring re-engineering of existing code and awkward constructions in future code.

Dangling pointers to collected objects wouldn't be created if `delete` didn't allow the storage occupied by collected objects to be reused immediately. However, we think garbage collection will be much more palatable to product engineers if they have the option of optimizing resource-critical parts of their systems via explicit `delete`'s that free storage immediately (see section 4.2.).

6.4. Type validity

Our definition of GC-safety allows a collected pointer of type `T*` to be stored in a variable of another type `U*`, even if `T` is not derived from `U`. GC-safety requires only that collected pointers be stored in variables, members, and elements declared to have some pointer type.

This weak type validity imposes one requirement on collectors: collected storage returned by `new` must always be aligned to the largest possible alignment of any type (typically 32- or 64-bit alignment), regardless of the actual type being allocated. This requirement ensures that if an address is truncated by an assignment of a `T*` into a `U*` (such as on a word-addressed machine), the truncated address still refers to the original collected object.

Most, perhaps all, implementations of `new` and `malloc` already behave this way. Though the *ARM* is silent on the issue, returning maximally aligned storage is required to implement overloading of `operator new`.

A stronger notion of type validity would require that a variable of type `T*` contains a pointer to a value of type `T` or a class derived from `T`. We know of only one garbage-collection technique that benefits from strong type validity. This technique uses the declared types of pointers to determine the types of objects in the heap rather than tagging the objects themselves, saving the space overhead of the tags. But this technique doesn't fully extend to languages with class inheritance—objects of class types still need to be tagged. As far as we know, this technique has never been implemented, and its space savings for typical C++ programs would be negligible.

Further, like pointer validity, strong type validity would restrict compatibility with existing C++ code.

7. Safe subset

Experience with other garbage-collected languages shows that when the program violates the GC-safety rules, the collector can arbitrarily scramble the heap. Thus, many researchers think it is quite important that languages and implementations automatically enforce the safe-use rules. While we ourselves don't think automatic enforcement is necessary for C++ garbage collection to be useful, we do think programmers should have the **option** of using automatic enforcement. To that end, we've designed a *safe subset* of C++.

The safe subset is just that: a true subset of the C++ language [Ellis 91], requiring no extensions or changes. Programmers are assured that code written in the safe subset is GC-safe, that is, it follows the safe-use rules of the garbage collector. More importantly, they are assured that code written in the safe subset cannot be responsible for storage-related bugs caused by dangling pointers or references, memory smashes, null-pointer dereferences, or invalid array indices. Programmers mark safe code with a pragma, and the compiler ensures that such code uses only the safe subset. It also generates some run-time checks to ensure safe use of particular language features—a program attempting to use an invalid pointer will halt with an error. Code written in the safe subset can be ported without change to C++ implementations that don't provide the subset.

Programmers need not write in the safe subset to use garbage collection, but then the responsibility is on them to ensure GC-safety and to avoid storage bugs. Use of the subset is not all-or-nothing. Even when some parts of a program can't be written in the safe subset, using it in the rest of the program reduces the potential for mistakes—when tracking down storage bugs, the programmer can safely rule out all code written in the subset.

The run-time checks are designed to have fairly low overhead so that they can be used throughout development and even in production. However, programmers can disable them at any time, trading safety for efficiency.

The safe subset ensures portability among different implementations of garbage collection by enforcing implementation-independent safe-use rules. An unsafe program may work with some collectors but not others; for example, an unsafe program may work with a mark-and-sweep collector but break with a copying collector.

The design of the safe subset is based on long experience with languages like Cedar, Modula-2+, Modula-3, and Ada [Rovner 85a, Rovner 85b, Nelson 91]. The safe subsets of these languages are expressive enough for applications and all but the lowest-level systems code and run-time facilities (such as device drivers). This C++ safe subset is noticeably less restrictive than the subsets of those languages, however. In particular, a non-collected object can point at a collected object, a pointer may address the interior of another object, and pointers may be freely passed to libraries written in other languages.

The full version of this proposal describes the safe subset in more detail [Ellis 93].

8. Suitable collection algorithms

The language interface to garbage collection (sections 4. and 5.) is compatible with all the major families of collection algorithms. But as a practical matter, some algorithms will be more suited for C++ garbage collection than others.

Practical collection algorithms for C++ must satisfy traditional requirements of garbage collectors:

> *Latency:* interruptions of the program should be short, less than 0.1 second for interactive programs.
>
> *Efficiency:* the collector should be time- and space-efficient, competitive with current implementations of `malloc/new`, and should be able to handle very large heaps.
>
> *Concurrency:* the collector should support multi-threaded programs running on multi-processors.

By "competitive", we mean only that programmers will find that they can achieve their goals more cheaply by using garbage collection. Even if collection were more expensive than explicit deallocation (and recent measurements indicate it often isn't [Zorn 92, Zorn 93]), many programmers would gladly pay that cost to increase productivity and decrease bugs.

Compared to previous collected languages like Lisp or Cedar, our design for C++ garbage collection imposes some additional, tougher requirements:

> *Interior pointers:* pointers can address the interior of objects. Traditional collectors support pointers only to the base of objects.
>
> *Cross-heap pointers:* non-collected objects can point at collected objects, and vice versa. Traditional collectors assume the language prohibits storing collected-heap pointers in the non-collected heap.
>
> *Unions containing collector pointers:* pointers to collected objects can be stored in union members. Unlike previous collected languages, C++ doesn't tag its unions.
>
> *Multi-lingual compatibility:* pointers to collected objects can be freely passed to libraries written in other languages, and those libraries' objects can point at collected C++ objects.
>
> *Minimal changes to compilers:* the more special support needed from the compiler, the less likely C++ garbage collection will be accepted and provided by vendors. Some algorithms, such as reference counting, require special language, compiler, or hardware support that wouldn't be practical in today's commercial multi-lingual environments.

In the last several years, researchers have evolved a class of collection algorithms meeting these requirements. Boehm et al. have developed mark-and-sweep collectors [Boehm 91], and Bartlett et al. have developed copying collectors [Bartlett 89, Detlefs 90, Yip 91], and the performance of these collectors is good enough for many C++ applications [Yip 91, Zorn 92, Zorn 93]. Both families of collectors use similar technology for satisfying our requirements.

Low latency (short interruptions) is achieved using virtual-memory synchronization to implement generational and concurrent collection [Shaw 87, Ellis 88].

Interior pointers are handled in the Boehm collectors by ensuring that all objects on a page have the same size and by using a table to map page numbers to object sizes. The Bartlett collectors use a bitmap per page indicating the starting offsets of objects.

Cross-heap pointers from the non-collected heap to the collected heap can be handled efficiently with the same VM-synchronization techniques used for generational collection. The non-collected heap is considered part of the root-pointer set of a generational collection, along with global data and the old-space pages of the collected heap. At the start of each generational collection, all the pages in the root set are write-protected, and the subsequent write-protection faults caused by the program tell the collector which of those pages have been modified; the collector needs to scan just those modified pages, not the

entire root set. The cost of this approach for a non-collected heap of N bytes and a collected heap of C bytes is about no worse than for a totally collected heap of $N + C$ bytes.

Most of today's major commercial platforms provide the basics for supporting virtual-memory synchronization, including Windows, Windows NT, OS/2, Macintosh, the various flavors of Unix, and almost all of the newer hardware for those systems. The collectors must be able to protect and unprotect data pages, intercept the resulting page-protection faults, and resume the program. The Boehm mark-and-sweep collectors only need write protection, whereas the Bartlett copying collectors need both write and no-access protection. The actual facilities provided by the various platforms may well be an imperfect match for the collectors' needs, and in particular, they may not be very well tuned [Appel 91], but experience to date indicates they are adequate for initial use by C++ collectors. Once paying customers start using C++ collectors, we can expect operating-system vendors to pay more attention to the requisite underlying facilities.

Multi-lingual compatibility and minimizing changes to C++ compilers dictate that the collector must be at least partially conservative. A totally conservative collector treats each word in a register, stack frame, or object as a possible pointer, whereas a totally precise collector knows exactly where each pointer is stored at all times. The Boehm mark-and-sweep collectors are totally conservative, while the Bartlett copying collectors scan the registers and stacks conservatively and the heap precisely. (Copying collectors must scan the heap precisely.)

Conservative scanning requires no compiler support and very little run-time support; the collector just needs access to the stacks and registers. With conservative collectors, libraries written in C and other languages can allocate collected objects without requiring changes to those compilers. Because each word in the registers, stacks, and heaps is treated as a pointer even if it isn't, conservative scanning sometimes retains unused objects accidentally, increasing memory usage.

Precise scanning requires a fair bit of compiler and run-time support (but no language changes). The compiler generates "type maps" describing the locations of pointers within objects, and `new` tags each object with a type map.

Most researchers believe precise scanning of the heap is more time- and space-efficient than conservative scanning, since it examines fewer words, does less work to identify true pointers, and doesn't accidentally retain unused objects. On the other hand, interpretation of type maps entails a fair bit of overhead. No one has yet published good comparisons of the two techniques. However, measurements of the Boehm collectors show that fully conservative scanning yields practical collectors [Boehm 91, Zorn 92, Zorn 93]. Boehm has recently developed simple techniques for dramatically reducing accidental retention of objects by conservative scanning [Boehm 93].

While it is technically possible to scan stacks and registers precisely, doing so without compromising code quality is complicated [Diwan 92], and it would be infeasible to modify all the other major language compilers as well as C++ compilers. The performance benefit of precise stack scanning is not large, since the average size of stacks is small and scanning them takes only a small fraction of total collection time.

There's been quite a bit of positive experience with algorithms that scan the stacks and registers conservatively and scan the heap precisely [Rovner 85a, DeTreville 90b, Bartlett 89]. However, those collectors didn't allow interior pointers; once interior pointers are allowed, there's a greater chance that a random word on the stack could be mistakenly interpreted as a true pointer and more unused storage retained. In practice, this doesn't appear to be a significant problem [Boehm 93].

The best C++ collectors might scan the stacks conservatively and use both precise and conservative scanning of the heaps. Objects allocated by C++'s `new` would be tagged with their type map and scanned precisely; objects allocated by C's `malloc` would be untagged and scanned conservatively.

As discussed in section 9., the language interface explicitly allows unions to contain collected pointers. This requires a collector without hardware support to scan pointer-valued union members conservatively, even if the collector uses otherwise totally precise scanning. Handling unions is straightforward for both conservative mark-and-sweep and mostly copying algorithms, which already have the necessary mechanisms to handle ambiguous pointers in the root set.

Detlefs hypothesized that it isn't possible to scan unions conservatively in a concurrent, mostly copying collector [Detlefs 90]. But we now believe that is wrong. A collector could maintain the set of all objects having pointer-containing unions (the set could be implemented as a threaded list). Before scanning the

heap, the collector marks all objects that are referents of those pointer-containing unions. If the collector later discovers that a marked object is actually reachable from the root set, it promotes the object rather than copying it (since it may be referenced by an ambiguous pointer). In a VM-synchronized collector, it's easy to do the initial marking from pointer-containing unions concurrently.

9. Unions

Our proposal explicitly allows unions containing pointers to collected objects, even though they can cause problems for garbage collectors. However, programs that make heavy use of pointer-containing unions may cause collectors to perform sub-optimally, and programmers must be warned to avoid such unions wherever possible. For new code, the proposal includes a standard template class `Variant` that provides programmers with an efficient alternative to unions. The rest of this section examines these issues.

9.1. The problem with unions

Unlike the variant records of Pascal or Modula, C++ unions are untagged, containing no indication of their current contents. Consider the union

```
union {int i; char* p;} u;
```

A collector can't tell whether u contains an integer or a pointer, since on most implementations the representations of integers and pointers are indistinguishable. These *ambiguous pointers* cause two problems for collectors.

First, conservative collectors simply assume that the union contains a pointer, whether it does or not, and retain the object that is apparently referenced by the ambiguous pointer value. This can cause excess unused storage to be accidentally retained. If pointer-containing unions are used extensively with a partially conservative mostly copying algorithm, the excess storage may be unacceptably large [Detlefs 90].

Second, pure copying collectors are in a bind: If the union contains a pointer, its referent must be moved and the pointer updated, but if it contains an integer, its value must be left unchanged. Without special hardware, untagged unions seem to preclude pure copying collectors.

9.2. Design alternatives

We considered three alternative solutions for coping with unions:

> disallow unions containing pointers to collected objects;
> allow pointer-containing unions, but change their representation to use implicit tags; or
> allow pointer-containing unions, with no change to their representation.

The first alternative, disallowing pointer-containing unions, would sidestep the performance problems of collectors, but it would add a restriction to the language and sacrifice compatibility with existing code and programming practice.

The second alternative allows pointer-containing unions but requires compilers to add implicit tags identifying their current contents to the collector. But appendix A shows it isn't feasible to tag unions while maintaining C++ semantics. Even if it were, the tagging would increase the size of unions, making their representation incompatible with data structures defined externally by hardware, file formats, and modules written in other languages like C. This would defeat our original purpose for allowing unions, maintaining compatibility with existing code.

The third alternative, allowing pointer-containing unions without changing their representation, maintains compatibility but imposes constraints on collectors. Unions must be scanned conservatively, precluding pure precise collectors and sometimes causing excess storage to be retained. Programmers must be warned of the possible increase in storage. Note, however, that a collector with otherwise totally precise scanning will incur no performance penalty if the program avoids pointer-containing unions.

In the end, we decided that avoiding language restrictions and maximizing compatibility were most important. Warning programmers to avoid pointer-containing unions is not ideal, but the

alternatives—outlawing unions entirely or making them incompatible with external representations—seem worse.

9.3. Variant template class

Though the proposal requires collectors to work correctly with pointer-containing unions, we recognize that they may be much less efficient if there are many ambiguous pointers. The language interface thus includes a standard `Variant` template class that provides programmers with the functionality of tagged unions:

```
template <class T1, class T2> Variant2 {
    int tag;
    union {T1 t1; T2 t2;} u;
    // public interface to Variant class...
    }
```

For example, `Variant2<int, char*> v` declares `v` to be a tagged union of an `int` and a pointer. A `Variant`'s tag is automatically set when a value is assigned to it, and it is optionally checked when the `Variant`'s value is accessed. By using `Variants` wherever possible, programmers can avoid the collector inefficiencies of untagged unions. The complete interface is presented in the full version of this proposal [Ellis 93].

A compiler/collector implementation may be built with particular knowledge about `Variants`. When it encounters a `Variant`, it can use the tag field to determine its current value and thus scan it precisely. Of course, conservative collectors can handle `Variants` correctly without any extra implementation work, though they won't receive the performance benefit.

Finally, note that `Variants` impose no further requirements for GC-safety. Since `Variants` are implemented with unions, the rules for unions apply equally to `Variants`. In particular, a GC-safe program must access only the currently assigned member of a `Variant`.

10. Interior pointers

Unlike most other languages with garbage collection, this proposal for C++ garbage collection allows *interior pointers*, pointers addressing the middle of objects. (Taking the address of an array element or an object's member yields an interior pointer.)

Language designers typically assume that interior pointers make garbage collection significantly more expensive. If interior pointers are allowed, a collector cannot assume that every pointer addresses the base of an object, and it must map every pointer it follows to its corresponding base. Since that mapping occurs in the collector's inner loops, it might cost a fair bit extra. Also, allowing interior pointers can cause conservative collectors to retain more storage.

Our initial design of the C++ safe subset prohibited interior pointers. But we eventually decided that it would be better to allow interior pointers, for several reasons.

First, to appeal to the largest number of programmers, we wanted to minimize restrictions on the language. Though practical systems programming languages like Cedar, Modula-3, and Ada prohibit or discourage interior pointers, we wanted to avoid antagonizing the many C++ programmers who would surely object to their absence.

Second, multiple inheritance creates problems not faced by other languages. Given a class `C` derived from classes `A` and `B`, in current implementations widening ("up-casting") a `C*` to a `B*` can create an interior pointer to the storage for `B` inside the `C` object. Thus prohibiting all interior pointers would require radical changes to current implementations of multiple inheritance. (The full version of this proposal [Ellis 93] presents a run-time representation of objects that could efficiently handle the special case of interior pointers resulting from widening casts, but this scheme would require non-trivial changes to current implementations that we think would impede acceptance of C++ garbage collection.)

Third, two recently developed families of partially conservative collection algorithms can handle interior pointers fairly cheaply [Bartlett 89, Yip 91, Boehm 91, Zorn 92, Zorn 93], and we believe that these algorithms best meet all the other practical requirements of C++ garbage collection (see section 8.).

We think that allowing general interior pointers will typically cost less than a few percent extra in total program execution time. (See below.)

Samples's language proposal [Samples 92] would facilitate more efficient following of interior pointers by the collector. Implementations could wrap each collected object, member object, and sub-object with a `gc wrapper` that would let the garbage collector quickly find the outermost containing object in the heap. Most likely, the `gc` wrapper would require no extra space in objects with virtual functions, but objects without virtual functions would need an extra header word in their representation. The cost of following interior pointers is only reduced, not eliminated, because the proposal still allows the declaration of `embedded` objects and pointers to such objects, with the intention that embedded objects don't have `gc` wrappers; following an embedded pointer would entail the full cost of mapping an interior pointer to the corresponding object base. As discussed in section 3.5., Samples's proposal doesn't meet our criteria of minimal changes and coexistence.

10.1. The run-time cost of interior pointers

The Boehm and Bartlett family of collectors provide heap data structures that allow the collector to map interior pointers to base pointers. The Boehm collector carves the heap into pages, with all of the objects on a page being the same size ("big bag of pages") and a page table giving the size of objects on each page. The Bartlett collectors keep a bit vector per page indexed by word offset, indicating the beginnings of objects on the page. In either scheme, mapping a pointer to a base pointer involves extracting a page index, indexing into the page table, then indexing another table or a bit vector.

On the MIPS processor, this mapping takes about 15 instructions, compared to zero instructions for a language like Modula-3 that prohibits interior pointers. Pointer following is the most frequent operation of a collector, and it must pay this 15-instruction penalty for every pointer it follows.

To get a feel for the magnitude of this penalty, we measured a commercial C++ debugger modified to use version 2.3 of Boehm's collector. Running on a 40 MHz DECstation 5000/240, we drove the debugger with a test script that executed its basic facilities:

> The script executed for 11 minutes, 20 seconds.
> The heap grew to 7.08 million bytes.
> There were 22 collections of the entire heap, taking an average of 1.48 seconds per collection.
> The collector followed 2.68 million pointers to actual objects.

Assume that 10 of the 15 instructions per pointer followed are devoted to handling interior pointers [Ellis 93], and pessimistically assume the processor executes 20 million instructions per second. Then only 0.2% of total execution time was spent handling interior pointers, which is about 0.061 seconds per collection or about 4% of the time spent in the collector.

Based on this one experiment, we think interior pointers are quite affordable. At worst, if the typical application spent 5 times as much time handling interior pointers, it would still cost just 20% of total collector time.

However, even if we outlawed interior pointers in the language, it would still be hard to eliminate all of the interior-pointer overhead from the Boehm and Bartlett family of collectors. As discussed in section 8., most implementations in the next few years will probably scan the stacks and registers conservatively, and such scanning needs a way to determine whether an address points at a heap object. The heap structures needed for this determination are very similar to those needed for handling general interior pointers, and the heap allocator must still maintain those structures.

Further, with mark-and-sweep collectors, it's generally more efficient to store object mark bits on the side to increase locality of reference and reduce dirtying of pages during the mark phase of collection, and it's required for concurrent VM-synchronized collectors. Storing mark bits on the side requires heap structures similar to those needed for interior pointers, and the instruction sequence for following a pointer is about the same regardless of whether interior pointers are allowed or not—a page index must be extracted from the pointer and used to index a page table. For example, version 2.3 of Boehm's collector can be configured to disallow interior pointers, yet the sequence for following a pointer is only one instruction shorter.

10.2. The space cost of interior pointers

Interior pointers have no effect on the space efficiency of a totally precise collector, but they can cause conservative collectors to retain more storage. When interior pointers are allowed with a conservative collector, a word is considered to point at an object if it points anywhere within the object, not just at its first byte. This increases the probability that a random bit pattern in a word will be misinterpreted as a valid pointer, and thus more objects may be unnecessarily retained. To the extent that a collector uses precise scanning, fewer words will be misinterpreted as interior pointers.

Fortunately, unnecessary retention isn't a serious problem with totally conservative collectors on newer architectures, especially with Boehm's techniques for improving space efficiency [Boehm 93]. The more limited experience reported for the Bartlett family of mostly copying collectors, which also allow interior pointers but scan the collected heap precisely, indicates that unnecessary retention is not a big problem with those collectors either [Detlefs 90, Yip 91].

11. Code-generator safety

Existing compilers may sometimes generate code incompatible with the correct operation of garbage collectors. Given an expression dereferencing the last use of a pointer `p`, many compilers may generate code that overwrites `p`'s register with a temporary address that points outside the bounds of the referenced object. If a concurrent collection occurs at that point, the object may get collected prematurely before the dereferencing expression completes.

A garbage collector determines which objects are no longer being used by tracing out the graph of reachable objects, using the registers, stacks, and static data segments as roots of the trace. Any object not traced is garbage and can be reused. The algorithms discussed in section 8. consider an address to point at an object if it points at the base of the object or anywhere inside it.

As long as the program thinks an object is live, the code generator should ensure there is at least one reachable pointer pointing at it; otherwise the collector may collect it prematurely. Consider this code fragment:

```
char* a = new char[10];
int i = 20, j = 19;
...a[i - j]...
```

If the expression `a[i - j]` is the last use of `a` (`a` is dead after the expression), then a code generator might decide to generate the following MIPS R3000 code:

```
    # a is in register $a, i in $i, j in $j
    # result will be placed in $r
addu $a, $a, $i
subu $a, $j
lb   $r, 0($a)
```

After the first instruction, register `$a` points at `a + 20`, that is, beyond the end of the object `a`. If there is no other pointer addressing the object, and if a garbage collection occurs at this point, the object would get garbage collected prematurely. (Concurrent garbage collectors can run at any time.)

Reduction of strength of loop induction variables can cause similar problems. For example, an optimizing compiler might transform this source:

```
char* a = new char[10];
for (i = 10; i < 20; i++) {
    ...a[i - j]...};
/* a is dead here */
```

into the equivalent of:

```
char* a = new char[10];
char* p = a + 10;
char* end = p + 20;
/* a is dead here */
for (; p < end; p++) {
    ...*(p - j)...}
```

Inside the loop, p always points past the end of the array a. The code generator or the calling sequence may cause the register containing a to be reused, leaving no pointer that points at or into the object and allowing it to be collected prematurely.

This problem is not academic—many commercial compilers perform these sorts of optimizations. Users of the current Boehm and Bartlett collectors are responsible for avoiding such situations themselves, and some conservative programmers disable optimization entirely. Though the problem appears to be rare in practice, it needs a robust solution.

11.1. A solution

In this section, we'll sketch a solution to code-generator safety that's appropriate for collectors that scan the stacks and registers conservatively, interpreting each word as a possible interior pointer.

Let p be a pointer-valued source expression, and let e be a dereferencing expression of the form p[i], *p, or p->f. The compiler must ensure that throughout the evaluation of e, some reachable pointer (possibly in a register) points at or into *p. The compiler can meet this constraint by extending the lifetime of p's value in the generated object code to include every load and store to an address derived from p. For example, if p is in a register, the compiler couldn't reuse that register until a load from an address derived from p completes.

Let's look at how a traditional compiler might implement this, assuming it generates code for one function at a time and does no inter-procedural optimization. Define a *base pointer* in the compiler's intermediate code to be a pointer-valued variable with no reaching definitions or the pointer-valued result of a function call or load, and a *derived pointer* to be any pointer resulting from address arithmetic on a base pointer or another derived pointer. Intermediate address, load, and store operations can be annotated with the base pointers from which their arguments are derived. For example, consider this intermediate code for the source expression a[i - j]:

```
t1 = a + i, {a}
t2 = t1 - j, {a}
x  = load t2, {a}
```

The address temporaries t1 and t2 and the load are derived from the base pointer a. The annotations must in general be sets, since a derived pointer may have multiple reaching definitions:

```
if e
    t1 = a, {a}
else
    t1 = b, {b}
t2 = t1 + i, {a, b}
x  = load t2, {a, b}
```

The definition of t2 has two reaching definitions for t1, so it is annotated with {a, b}.

The compiler's live-variable analysis can treat the annotated base pointers of loads and stores as uses of those pointers, effectively extending the live ranges of the base pointers to include the loads and stores. Once the loads and stores have been annotated with their bases, the annotations of address temporaries can be discarded, and the annotations of the loads and stores will remain valid even after traditional optimizations. The compiler must be careful to ensure that the extended live ranges of base pointers are observed by the later phases of register allocation, instruction selection, scheduling, and peephole optimization.

Suppose the source program creates a derived pointer pointing outside of its referenced object:

```
char* a = new char[10];
a = a + 12;
...
c = a[-12];
```

The behavior of such programs is explicitly undefined under the ANSI C standard and the draft C++ standard, so the compiler is not obligated to maintain code-generator safety. It can assume that a pointer-valued source expression points at or into the object referenced by the expression's original base pointer.

Extending the lifetime of a base pointer p isn't required if all pointers derived from it are guaranteed to point into *p. Consider this example:

```
struct S {int i; char a[10];} *s;
...s->a[i]...
/* s is dead here */
```

A valid (but suboptimal) code sequence for the R3000 is:

```
addiu $s, $s, 4     /* add the offset of a */
addu  $s, $s, $i    /* index into a with i */
lb    $r, 0($s)     /* load the byte */
```

This sequence is safe, since $s always points at or into the object *s.

Extending the live ranges of base pointers has little impact on the quality of code generated for modern RISC architectures. In the common cases, either p is live at the end of all its dereferencing expressions, or else the temporary pointers created by the expressions point into *p. Only when p is dead at the end of a dereferencing expression must the code generator consider extending its live range. For a single expression based on p, this may cause p's register to be retained for a few extra instructions, and only in rare situations would this cause a register spill. For an optimized loop, p's register could either be retained for the duration of the loop or, if there aren't enough registers, stored in a stack temporary at the beginning of the loop.

Safety is easy to implement in a code generator written from scratch. We used this approach several years ago to add GC-safety to a better-than-pcc-quality Modula-2+ compiler implemented at SRC, and it took less than 50 lines of additional code. Obviously, retrofitting an existing optimizing compiler could be harder, though it should be only a small part of the total cost of adding garbage collection to a C++ implementation.

Our approach to code-generator safety is based on that described by Boehm [92], but somewhat simpler. Their main concern was how to use C as an intermediate code for other compilers, so they worried about handling source not conforming to the ANSI standard. They also assumed that target garbage collectors might not handle interior pointers, requiring more careful handling of base pointers by the compiler.

Diwan et al. describe an approach suitable for totally precise copying collectors [Diwan 92]. Their scheme is considerably more complicated to implement in the presence of optimization, but it allows collectors to relocate any object. The approach described here, since it considers the stacks and registers to contain ambiguous roots, requires the use of partially conservative collectors such as Bartlett's mostly copying algorithm.

Finally, researchers working on smart pointers have tried to provide code-generator safety purely at the source level without modifying the compiler [Detlefs 92]. In general, this isn't possible without increasing the natural size of pointers, relying on details about a particular implementation's compiler, or relying on the vaguely specified, inefficient `volatile` type attribute.

11.2. Unsafe libraries

Our proposal for C++ garbage collection emphasizes coexistence with existing non-collected libraries written in C++ and other languages. In particular, we feel it's very important to allow collected objects to be passed to such libraries.

Unfortunately, it is unlikely that all or even most such libraries will be compiled with safe code generators in the near future. A programmer who writes in the safe subset and uses a safe C++ compiler may still have problems if he must use a library compiled with an unsafe compiler.

Luckily, experience with the Boehm collectors indicates that in practice, lack of safely compiled libraries may not be a serious problem. Objects being manipulated by a function almost always have a base pointer stored somewhere, either in the heap or in a caller of the current function. Further, since objects created and managed by the existing libraries will be in the non-collected heap, the only vulnerable collected objects are those created by clients and passed to the libraries as uninterpreted "client data". Since the libraries view such objects as `void*` pointers, they won't be dereferencing them, and unsafe addressing expressions won't be executed.

Obviously, conservative programmers won't be completely satisfied with assurances that problems will be "rare". Ideally, we would give them 100% confidence. Until garbage collection is accepted as an indispensable tool, however, many vendors will see little need to provide safe compilers and safely compiled libraries, especially for languages other than C++. But even without safely compiled libraries, we think garbage collection will greatly decrease the total cost of storage bugs; we think most programmers would rather deal with extremely infrequent bugs caused by unsafe libraries than with the very frequent storage bugs and design problems they see today without garbage collection.

12. Standardization

Before C++ garbage collection can be widely used, there must be a standard that vendors and programmers can rely on. Programmers want assurance they're not tied to a particular vendor or platform, and vendors want assurance that programmers won't view garbage collection as a vendor-specific language extension. Initially, the standard will most likely be a de facto agreement among the few adventurous vendors who might risk providing garbage collection to their users. But eventually, garbage collection will have to be included in the ANSI standard.

12.1. What should be included

The standard should include the following components:

the language interface (section 4.);
the specification of object clean-up, including the standard interface `WeakPointer.h` (section 5. and appendix B);
the definition of GC-safety (section 6.).

The safe subset is not required to use garbage collection, but for those programmers who want to use the safe subset, its definition should be standardized to allow portability.

12.2. What should be excluded

As part of our system-level approach to adding collection to C++, we've discussed quite a few implementation issues. While implementation issues are important to consider when designing a programming language, the final language design should leave implementors as much freedom as possible to best meet the needs of their customers.

The following implementation issues should be explicitly excluded from a standard:

requirements for specific algorithms;
how C++ garbage collection interacts with other languages;
performance specifications of garbage collection;
how to implement code-generator safety.

The standard should not specify particular collector algorithms. No single algorithm is ideal for all programs, and we want to allow vendors and researchers as much flexibility as possible for designing new algorithms. The GC-safety rules let programmers write portable programs without relying on particular collector implementations.

In general, a C++ language standard can't specify how C++ should interact with other languages, since it doesn't have control over those languages. Those languages weren't designed or implemented with collection in mind, so there is no way to ensure that implementations of those languages will be compatible with particular implementations of collection. Our proposal allows straightforward collector implementations that should be compatible with many common language implementations. But the degree of compatibility with other languages must necessarily depend on which collector algorithm has been chosen by the C++ vendor and on the implementations of the other languages. C++ vendors (particularly those who also offer C compilers) may choose to sacrifice some general compatibility in favor of more efficient collection algorithms.

Performance specifications of garbage collection (that is, space and time usage) should not be included in the standard, just as the current draft standard does not specify the performance of `new` and `delete` (or any other part of the language, for that matter). To date, no one knows how to write such a specification that is helpful to programmers, feasible to implement, and flexible enough to allow different implementations of storage management.

Finally, as with all issues of compiler implementation, how code-generator safety is implemented should not be included in the standard.

13. Summary of implementation changes

Here's a summary of the changes that must be made to current C++ implementations to support this proposal.

The language interface requires:

providing the `gc` keyword,
changing `new T` to call the collector's allocator when `T` is a `gc` type,
changing `delete e` to call the collector when `e` points at a collected object.

The safe subset requires:

providing `#pragma safe`,
adding a "safe" boolean attribute to names in the compiler's symbol table,
10 compile-time checks,
6 run-time checks,
2 compiler restrictions.

Both the language interface and the safe subset can be implemented as localized changes to the compiler's front end.

The compiler must be changed to provide code-generator safety.

The sources to the Boehm and Bartlett collectors are freely available for unrestricted commercial use.

14. Conclusion

For better or worse, use of C++ will surely increase over the next many years. Of all the different ways studied by researchers for improving programmer productivity, adding garbage collection to C++ could give a big bang for a small buck.

Researchers have been touting the virtues of garbage collection for three decades, and it's time for us to put up or shut up. To that end, we've presented a two-part proposal. The first part adds garbage collection to C++, and the second part defines an optional safe subset that enforces correct, portable use of garbage collection and precludes storage bugs. Though both parts are important, garbage collection by itself is so valuable that we think near-term efforts should focus on it, rather than the safe subset. Once collection is accepted, it will be easy enough to provide a safe subset for those who want it.

We've only started the long chicken-and-egg process of providing C++ programmers with safe garbage collection. C and C++ programmers tend to be conservative, and experience based on other programming languages doesn't impress them. (It took over a decade for static type safety to be accepted by C programmers.) Most commercial programmers won't use a new language tool until it's widely available on

several platforms, but most vendors are reluctant to offer new tools until there is a demonstrated demand for them and there is a corresponding standard. Standards committees are reluctant to standardize technology that isn't yet in wide use. There is no free lunch.

Acknowledgments

Al Dosser and Joel Bartlett helped us early on with the language interface.
Kelvin Don Nilsen and Jerry Schwarz prodded us into thinking more carefully about GC-safety.
Hans Boehm was a patient sounding board for new ideas.
Thomas Breuel sharpened our arguments with lively debate.
Dain Samples helped us understand his alternative proposal.
Bjarne Stroustrup provided helpful feedback on the likely evolution of C++.
Cynthia Hibbard edited a previous version of this paper.

References

[Appel 91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. *In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[Bartlett 89] Joel F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Western Research Laboratory Technical Report TN-12, Digital Equipment Corporation, 1989.

[Boehm 91] Hans-J. Boehm, Alan J. Demers, Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[Boehm 92] Hans-J. Boehm and David Chase. A proposal for garbage-collector-safe C compilation. *The Journal of C Language Translation* 4(2):126-141, December 1992.

[Boehm 93] Hans-Juergen Boehm. Space Efficient Conservative Garbage Collection. In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, 1993.

[CenterLine 92] CenterLine Software, Cambridge, Massachusetts. *CodeCenter, The Programming Environment*, 1992.

[Codewright 93] Codewright's Toolworks, San Pedro, CA. *Alloc -GC: The garbage collecting replacement for malloc()*, 1993.

[Detlefs 90] David L. Detlefs. Concurrent garbage collection for C++. CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, 1990.

[Detlefs 92] David L. Detlefs. Garbage collection and run-time typing as a C++ library. In *Proceedings of the 1992 Usenix C++ Conference*, 1992.

[DeTreville 90a] John DeTreville. Heap usage in the Topaz environment. Systems Research Center Report 63, Digital Equipment Corporation, 1990.

[DeTreville 90b] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Systems Research Center Report 64, Digital Equipment Corporation, 1990.

[Diwan 92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[Dix 93] Trevor I. Dix and Tam T. Lien. Safe-C for introductory undergraduate programming. In *The 16th Australian Computer Science Conference*, 1993.

[Edelson 91] D. R. Edelson and I. Pohl. Smart pointers: They're smart but they're not pointers. In *Proceedings of the 1991 Usenix C++ Conference*, April, 1991.

[Edelson 92] Daniel R. Edelson. Precompiling C++ for Garbage Collection. In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92*. Springer-Verlag, 1992.

[Ellis 88] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Systems Research Center Report 25, Digital Equipment Corporation, 1988.

[Ellis 91] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[Ellis 93] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for C++. Systems Research Center Report 102, Digital Equipment Corporation, 1993. Also published as Xerox Palo Alto Research Center report CSL-93-4, 1993.

[Ginter 91] Andrew Ginter. Design alternatives for a cooperative garbage collector for the C++ programming language. Research Report No. 91/417/01, Department of Computer Science, University of Calgaray, 1992.

[Hayes 92] Barry Hayes. Finalization in the collector interface. In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92*. Springer-Verlag, 1992.

[McJones 87] Paul R. McJones and Garret F. Swart. Evolving the UNIX system interface to support multithreaded programs. Systems Research Center Report 21, Digital Equipment Corporation, 1987.

[Pure 92] Pure Software, Los Altos, California. *Purify Version 1.1 Beta A*, 1992.

[Nelson 91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Owicki 81] Susan Owicki. Making the world safe for garbage collection. In *Eighth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[Rovner 85a] Paul Rovner. On adding garbage collection and runtime types to a strongly typed, statically checked, concurrent language. Xerox Palo Alto Research Center report CSL-84-7, 1985.

[Rovner 85b] Paul Rovner, Roy Levin, and John Wick. On extending Modula-2 for building large, integrated systems. Systems Research Center Report 3, Digital Equipment Corporation, 1985.

[Samples 92] A. Dain Samples. GC-cooperative C++. In Y. Bekkers and J. Choen, editors, *Memory Management, International Workshop IWMM 92*. Springer-Verlag, 1992.

[Shaw 87] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Computer Systems Laboratory, Stanford University, 1987.

[Steffen 92] Joseph L. Steffen. Adding run-time checking to the Portable C Compiler. Software—Practice and Experience 22(4):305-316, April 1992.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language, Second Edition.* Addison-Wesley, reprinted with corrections December 1991.

[Stroustrup 92] Bjarne Stroustrup and Dmitry Lenkov. Run-time type identification for C++ (revised). In *Proceedings of the 1992 Usenix C++ Conference*, 1992.

[Yip 91] G. May Yip. Incremental, generational, mostly-copying garbage collection in uncooperative environments. Western Research Laboratory Research Report 91/8, Digital Equipment Corporation, 1991.

[Zorn 92] Benjamin Zorn. The measured cost of conservative garbage collection. Technical Report CU-CS-573-92, Department of Computer Science, University of Colorado at Boulder, 1992.

[Zorn 93] Benjamin Zorn, David Detlefs, and Al Dosser. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-92, Department of Computer Science, University of Colorado at Boulder, 1993.

Appendix A: Why tagged unions aren't practical

Unions containing collected pointers pose special problems for copying collectors. For example, consider:

```
union {int i; char* p;} u;
```

How does the collector know whether u contains an integer or a pointer? A pointer must be relocated by the collector, and an integer must not. In other languages such as Cedar and Modula-2+, variant records have tags indicating their current contents, but in traditional C++ implementations, unions are untagged.

Tagging unions in C++ isn't practical for two reasons: it can't be done efficiently while conforming to C++ semantics, and it would sacrifice representational compatibility with external data structures.

Suppose that the compiler adds an implicit tag to each union and generates code to change the tag when a member is assigned. This works fine for simple assignments to members, but there doesn't seem to be any efficient method of updating the tag when a member is changed by assignment through an alias (a pointer or reference to the member). Consider this fragment, which conforms to the ANSI C and ARM union semantics:

```
union U {int i; char* p;};
U u;
char** ptrToP;

u.i = 0;              /* u contains an integer i */
ptrToP = &u.p;        /* create an alias to u.p */
... = u.i;            /* u still contains integer i */
*ptrToP = ...;        /* assign u.p via the alias */
... = u.p;            /* u now contains pointer p */
```

The union member `u.p` gets changed via assignment through the alias `*ptrToP`. In general, since almost any pointer could be an alias for a union member, there isn't any way a compiler could generate efficient code to maintain union tags.

Even if tagging could be done efficiently, the tags would increase the size of unions. Unions are frequently used to access data defined by hardware devices, external file formats, and modules written in other languages (especially C), and increasing the size of unions would make them incompatible with these external representations.

Appendix B: WeakPointer.h

This appendix presents the standard interface `WeakPointer.h`, which provides weak pointers and object clean-up.

```
#ifndef       _WeakPointer_h_
#define       _WeakPointer_h_

/*********************************************************************
```

WeakPointer and CleanUp

```
*********************************************************************/

#pragma safe

/*********************************************************************
```

WeakPointer

A *weak pointer* is a pointer to a heap-allocated object that doesn't prevent the object from being garbage collected. Weak pointers can be used to track which objects haven't yet been reclaimed by the collector. A weak pointer is *deactivated* when the collector discovers its referent object is unreachable by normal pointers (reachability and deactivation are defined more precisely below). A deactivated weak pointer remains deactivated forever.

```
*********************************************************************/


template< class T > class WeakPointer {
public:

WeakPointer( T* t = 0 );
      /* Constructs a weak pointer for *t. t may be null. It is an error if t is non-null and *t is not a
      collected object. */

T* Pointer();
      /* wp.Pointer() returns a pointer to the referent object of wp or null if wp has been
      deactivated (because its referent object has been discovered unreachable by the collector). */

int operator==( WeakPointer< T > );
      /* Given weak pointers wp1 and wp2, if wp1 == wp2, then wp1 and wp2 refer to the same
      object. If wp1 != wp2, then either wp1 and wp2 don't refer to the same object, or if they do, one
      or both of them has been deactivated. (Note: If objects t1 and t2 are never made reachable by
      their clean-up functions, then WeakPointer<T>(t1) == WeakPointer<T>(t2) if and
      only t1 == t2.) */

int Hash();
      /* Returns a hash code suitable for use by multiplicative- and division-based hash tables. If wp1
      == wp2, then wp1.Hash() == wp2.Hash(). */
};

/*********************************************************************
```

CleanUp

A garbage-collected object can have an associated clean-up function that will be invoked some time after the collector discovers the object is unreachable via normal pointers. Clean-up functions can be used to release resources such as open-file handles or window handles when their containing objects become unreachable. The initial clean-up function of a collected object is its destructor.

There is no guarantee that the collector will detect every unreachable object (though it will find almost all of them). Clients should not rely on clean-up to cause some action to occur—clean-up is only a mechanism for improving resource usage.

Every object with a clean-up function also has a clean-up queue. When the collector finds the object is unreachable, it enqueues it on its queue. The clean-up function is applied when the object is removed from the queue. By default, objects are enqueued on the garbage collector's queue, and the collector removes all objects from its queue after each collection. If a client supplies another queue for objects, it is his responsibility to remove objects (and cause their functions to be called) by polling it periodically.

Clean-up queues allow clean-up functions accessing global data to synchronize with the main program. Garbage collection can occur at any time, and clean-ups invoked by the collector might access data in an inconsistent state. A client can control this by defining an explicit queue for objects and polling it at safe points.

The following definitions are used by the specification below:

> Given a pointer `t` to a collected object, the *base object* `BO(t)` is the value returned by `new` when it created the object. (Because of multiple inheritance, `t` and `BO(t)` may not be the same address.)
>
> A weak pointer wp references an object `*t` if `BO(wp.Pointer()) == BO(t)`.

```
**********************************************************************/

template< class T, class Data > class CleanUp {
public:

static void Set( T* t, void c( Data* d, T* t ), Data* d = 0 );
      /* Sets the clean-up function of object BO(t) to be <c, d>, replacing any previously defined
      clean-up function for BO(t); c and d can be null, but t cannot. Sets the clean-up queue for
      BO(t) to be the collector's queue. When t is removed from its clean-up queue, its clean-up will be
      applied by calling c(d, t). It is an error if *t is not a collected object. */

static void Call( T* t );
      /* Sets the new clean-up function for BO(t) to be null and, if the old one is non-null, calls it
      immediately, even if BO(t) is still reachable. Deactivates any weak pointers to BO(t). */

class Queue {public:
    Queue();
       /* Constructs a new queue. */

    void Set( T* t );
       /* q.Set(t) sets the clean-up queue of BO(t) to be q. */

    int Call();
       /* If q is non-empty, q.Call() removes the first object and calls its clean-up function; does
       nothing if q is empty. Returns true if there are more objects in the queue. */
    };
};

/**********************************************************************
```

Reachability and Clean-up

An object `O` is *reachable* if it can be reached via a non-empty path of normal pointers from the registers, stacks, global variables, or an object with a non-null clean-up function (including `O` itself).

This definition of reachability ensures that if object `B` is accessible from object `A` (and not vice versa) and if both `A` and `B` have clean-up functions, then `A` will always be cleaned up before `B`. Note that as long as an object with a clean-up function is contained in a cycle of pointers, it will always be reachable and will never be cleaned up or collected.

When the collector finds an unreachable object with a null clean-up function, it atomically deactivates all weak pointers referencing the object and recycles its storage. If object `B` is accessible from object `A` via a path of normal pointers, `A` will be discovered unreachable no later than `B`, and a weak pointer to `A` will be deactivated no later than a weak pointer to `B`.

When the collector finds an unreachable object with a non-null clean-up function, the collector atomically deactivates all weak pointers referencing the object, redefines its clean-up function to be null, and enqueues it on its clean-up queue. The object then becomes reachable again and remains reachable at least until its clean-up function executes.

The clean-up function is assured that its argument is the only accessible pointer to the object. Nothing prevents the function from redefining the object's clean-up function or making the object reachable again (for example, by storing the pointer in a global variable).

If the clean-up function does not make its object reachable again and does not redefine its clean-up function, then the object will be collected by a subsequent collection (because the object remains unreachable and now has a null clean-up function). If the clean-up function does make its object reachable again and a clean-up function is subsequently redefined for the object, then the new clean-up function will be invoked the next time the collector finds the object unreachable.

Note that a destructor for a collected object cannot safely redefine a clean-up function for its object, since after the destructor executes, the object has been destroyed into "raw memory". (In most implementations, destroying an object mutates its vtbl.)

Finally, note that calling `delete t` on a collected object first deactivates any weak pointers to `t` and then invokes its clean-up function (destructor).

```
**********************************************************************/
#endif /* _WeakPointer_h_ */
```

Template Base Delegation

Ted Law
tedlaw@vnet.ibm.com

IBM Software Solutions Toronto Laboratory
844 Don Mills Road
North York, Ontario M3C 1V7
Canada

ABSTRACT

When a class derives from a template instantiation, the base class is called a template base class. Non-template base classes are more commonly used, but do not take into account the specific needs of the derived class. The base class can provide better service; it can be customized by becoming a template instantiation using the derived class and/or other classes as arguments. In this way, the template base class can tailor-make itself for the derived class, taking advantage of the knowledge of its type. For example, the template base class can type its methods appropriately for the derived class. Furthermore, inheriting publicly from a template base class makes it possible for the derived class to delegate part of its public type-specific interface to the base. This paper explores the advantages of template base delegation.

1 Introduction

In C++ object-oriented programming, objects frequently delegate responsibilities to other objects to enhance modularity, reuse, and extendability[5]. Responsibilities can also be abstracted into the base class to increase sharing among derived classes. Usually the type of the base class is fixed. Consequently, the signature of the methods and the types of the data members of the base class must be fixed. By turning the base class into a template instantiation, it can be customized according to the type of the derived class.

To illustrate, Fig. 1(a) shows two classes `D1` and `D2` inheriting from a common non-template base class `B`. Fig. 1(b) shows the technique of template base delegation, where `D1` and `D2` inherit from template base classes `B_<D1,...>` and `B_<D2,...>`, respectively.

This is a powerful technique that can be usefully applied to a variety of domain areas.

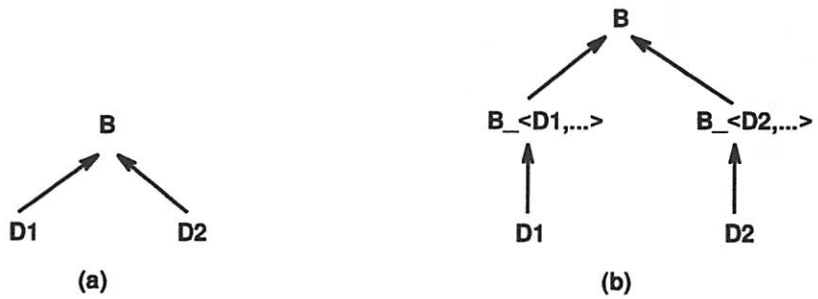


Figure 1: Using template instantiations as bases

Section 2 demonstrates how template base delegation can be used to provide an alternative to the container class implementation of lists. This simple example is intended for illustrative purposes. Section 3 builds on this example, explaining how the technique can be usefully applied to the area of object-modeling.

2 A Simple Example of Template Base Delegation

Many introductory text books on C++ use `List<T>` as an example for source code reuse[5]. `List<T>` is called a container class template.

A limitation of embedding a type `T` object in `ListItem<T>` is that the type `T` object can belong to at most one `List<T>`. This limitation can be readily overcome by using `List<T*>` instead of `List<T>`. See Fig. 2.

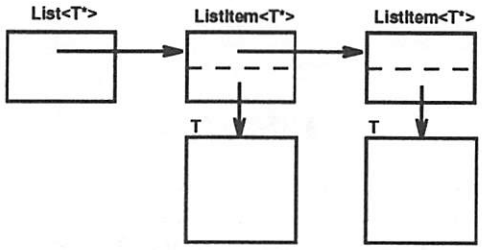


Figure 2: `List<T*>`

The number of `List<T*>`s that can contain the same `T` object can now vary dynamically. There is no predefined limit on the number of `List<T*>`s to which a `T` object can belong. The cost of this feature is the allocation cost of `ListItem<T*>` and the extra level of indirection to get to the `T` object. This cost is more expensive than we need because the number of lists that an object needs to be on is usually fixed at compile time.

Consider the ER diagram[1] in Fig 3, which shows the relationships in a `Company-Employee` example. In this example, `Manager` is a semantic *subtype* of `Employee`.

Assuming the two relationships are implemented using simple linked lists, Fig. 4 represents one possible runtime instance of some information.

In each `Employee` object there are two next pointers: one for chaining all the employees of the same company; the other for chaining all the employees under the supervision of the same manager, who himself is also an employee.

Since this application has two almost identical linked lists, there is an opportunity for source code sharing that can be realized by wrapping each next pointer in a base class. However, to distinguish between the two different next pointers, we can turn the base into a template called `FanTip_`[2][3] and parameterize it by the kind of the relationship that is being represented. Similarly, the head pointers at the beginning of the lists are wrapped into a template called `FanHead_`. Together they are called the Fan base class templates. For readers interested in more detail, the source code for this example is given in the Appendix.

[1] Entity-Relationship Modeling is a popular semantic data modeling technique in the database area. [5]

[2] The shape of a hand fan is used to name these templates. In the example, a `FanHead_` is the head of a simple linked list, and a `FanTip_` is an item on the list.

[3] The naming convention used here:

a. Data members are always postfixed by an underscore.

b. An underscore at the end of a class name indicates that the class is meant only to be used as a base of another class. It is not stand-alone.

c. For function parameters, references are used for objects that must exist, and pointers are used if they may exist.

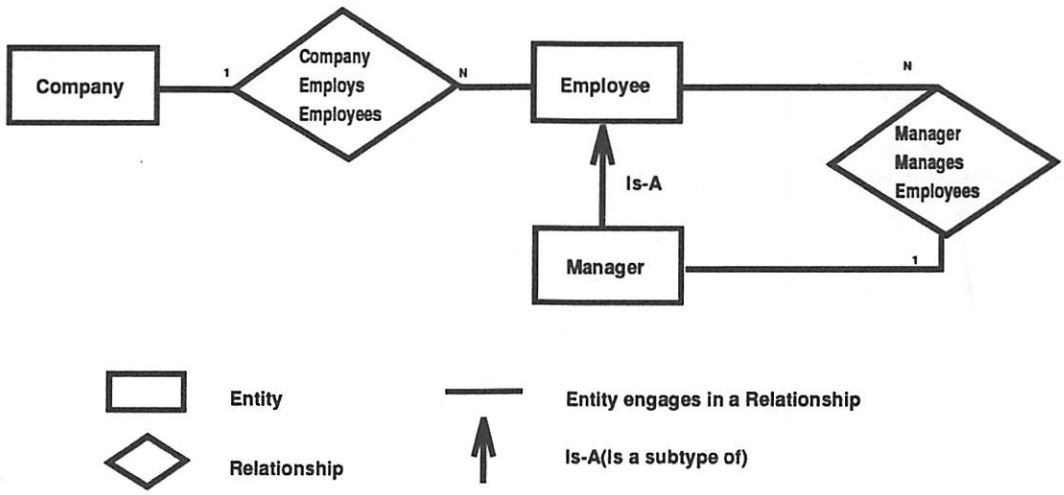


Figure 3: ER diagram of the Company-Employee example

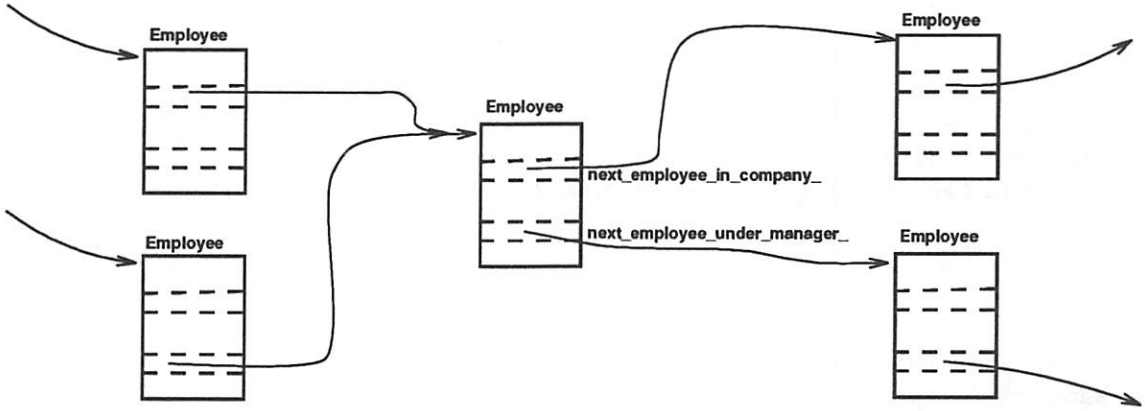


Figure 4: A runtime instance of multiple linked lists

Both `FanHead_` and `FanTip_` take three arguments: the type of the 1 side, the type of the N side, and the type of the 1-N relationship. The template class definitions are written so that the class definitions of the template arguments are not needed. In fact, the definitions of `CompanyEmploys Employees`[4] and `ManagerManagesEmployees` do not need to exist unless they are required for other reasons (for example, to carry an attribute of the relationship).

Since `FanHead_` and `FanTip_` know the type of the derived class, their interfaces can be typed accordingly. For example, `FanHead_` defines an `add_tip()` method, which takes a parameter of type `Tip&` (the second template argument). The client class `Company` customizes its base by inheriting from `FanHead_<Company,Employee,CompanyEmploysEmployees>`. Thus, the inherited `add_tip()` method will expect an `Employee&` as a parameter.

In addition to the advantages of source code sharing and strong typing discussed above, using the Fan template bases also makes the definitions of the derived classes more declarative. The template bases *declare* what relationships the derived class can engage in using the inheritance clause of a C++ class declaration:

[4] The convention is that the name of a relationship has the encoding, Noun1VerbNoun2, for example `Company EmploysEmployees`. The "direction" of the relationship is always from Noun1 to Noun2. Noun1 is always on the outgoing side, and Noun2, incoming.

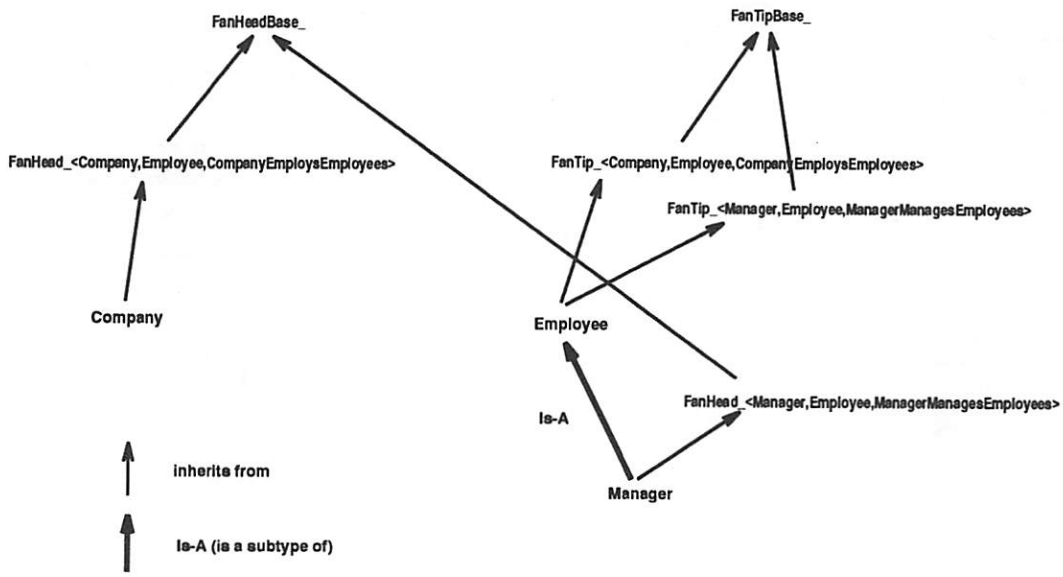


Figure 5: Inheritance hierarchy of the Company-Employee example

```
class Employee :
  protected FanTip_<Company,Employee,CompanyEmploysEmployees>,
  protected FanTip_<Manager,Employee,ManagerManagesEmployees>
{
    ...
};
```

Protected inheritance is used because `Manager`, a subtype of `Employee`, may need to access the methods which `Employee` inherits from its `FanTip_` bases. Public inheritance is not used in this case because `Employee` is not, semantically, a subtype of `FanTip_<Company,Employee,Company EmploysEmployees>`.

Typically, the binary code for every instantiation of a template is generated. To avoid generating an undue amount of binary code, we can make `FanHead_` inherit from a non-template base class `FanHeadBase_`, and `FanTip_` from `FanTipBase_`. There is no need for these non-template bases to realize the aforementioned advantages (source code sharing, strong typing, and declarative programming). However, factoring most of the code out of the template classes and into their non-template bases means that only one copy of the binary code needs be generated. After the bulk of the code is moved into these non-template bases, only pointer adjustment code, resulting from up/down casting, is left in the templates. When all the template code is inlined, the same efficiency can be obtained as any C equivalent. Thus storage and runtime efficiency can be added to the list of advantages of applying the technique of template base delegation to the problem of implementing linked lists.

If we compare the Fan templates with the conventional `List` container template, we will see that `List` makes no assumption about its client class, while `FanHead_` and `FanTip_` each assumes that it is inherited by its own client class. Using this assumption, `FanHead_` and `FanTip_` can embed, respectively, the `first_tip_` and the `next_tip_` pointers into objects of their client classes.

In summary, template base delegation allows a base class to be customized according to the derived class. That customization can be on the signatures of the methods or on the types of the data members. It gives the base class a chance to know exactly who is inheriting from it. Using template

bases is also more declarative, directing the reader of the source code to the actual intentions of the programmer. The example shown in this section is a simple application of the technique. The author has found this technique to be quite useful in the area of object-modeling, which is explained next.

3 A Real Life Example in Object Modeling

In many C++ database-like applications, there is a C++ class, called an *entity class*, used to represent each type of entities in the real world. And there is a C++ class, called a *relationship class*, used to represent each type of relationships in the real world. Furthermore, the application code needs to inquire into the possible relationships among these types of entities and their attributes at run time. This information is often called the (conceptual) schema of the application. To make the schema information available, we represent the schema at run time using a network of objects called *descriptors*. We use a unique *entity descriptor* object to represent each entity class, and a unique *relationship descriptor* object for each relationship class. A challenging problem is to ensure the consistency between the schema information as manifested in the source code and the explicit representation of that schema at run time.

Manually maintaining a schema generation routine or table according to the schema used in the source code is error prone. It is easy to change one but forget to update the other.

Consider the inheritance hierarchy shown in Fig. 6 for the same conceptual schema as was used in the previous example (Fig. 3).

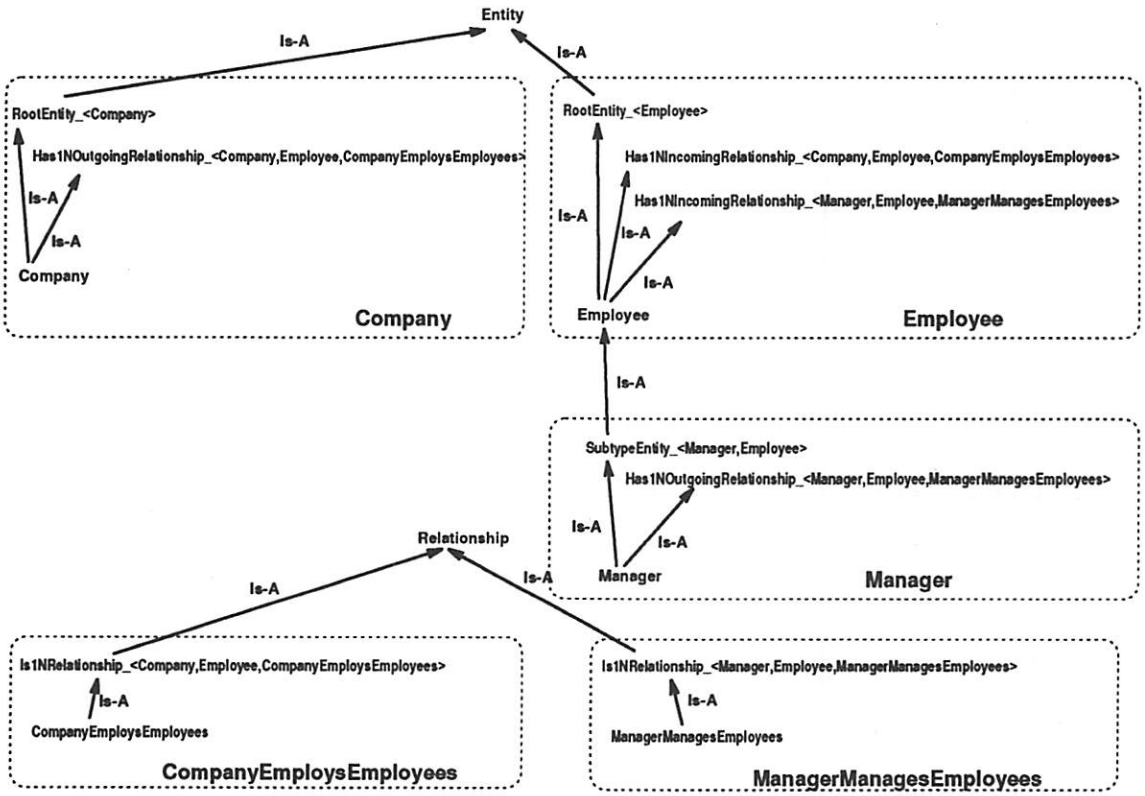


Figure 6: Inheritance hierarchy when using template bases to capture schema information

The fact that `Company` and `Employee` engage in a 1-N `CompanyEmploysEmployees` relationship is captured by `Company` inheriting from the base class `Has1NOutgoingRelationship_<Company, Employee,CompanyEmploysEmployees>`, and `Employee` inheriting from the base class `Has1NIncom ingRelationship_<Company,Employee,CompanyEmploysEmployees>`.

The fact that Manager is a subtype of Employee is captured by Manager inheriting from Subtype Entity_<Manager,Employee>. The template SubtypeEntity_ is written so that it derives publicly from its second template argument. This is a new kind of customization that allows the client to specify to the template base whom to inherit from, so that the intended subtype hierarchy in the application is preserved.

Every inheritance shown in Fig. 6 is public. Since the public interface of a class is the union of the public interface of all of its public bases, we can look at an entity class together with its public template bases as forming an "expanded" public interface for that semantic entity. This is shown by the dotted rectangles in the figure. In a way, each entity class have delegated part of its public interface to its template bases. Alternatively, each template base can be viewed as representing a particular role that the semantic entity can play. Note that, in the conceptual schema shown in Fig 3, there are three kinds of semantic entities, Company, Employee and Manager, where Manager is a subtype of Employee; and there are two kinds of relationships, CompanyEmploysEmployees and ManagerManagesEmployees. These facts will be captured by the template bases. Other helper classes can also be added to the hierarchy as required without affecting the representation of this schema.

3.1 Building the Schema Representation

Since template bases have been used to "declare" the various pieces of the schema, this information can be used to construct the schema's runtime representation automatically.

There is a static "init" object defined for each instantiation of RootEntity_, SubtypeEntity_, and Is1NRelationship_. Each init object uses the type information supplied to the template base to specify how a piece of the schema should be built.

The construction of the schema representation begins by chaining these "init" objects together to form a single list. This chaining can be easily achieved during static initialization by the constructor of a common base class, Init_, for these static init objects.

The rest of the construction process is carried out when a static member function Init_::initialize() is called sometime after main() is entered.[5] In this function, several phases may be involved. During each phase, a pass is made through the list of init objects, calling a different virtual method for each phase. After each phase is finished, the next phase begins, until all the phases are carried out.

There are two important phases.

The build_descriptor_phase is the phase for allocating all descriptor objects. For example, when the build_descriptor_phase virtual method of the init object of RootEntity_<Employee> is called, it will allocate an entity descriptor object to represent the entity class Employee. The build_descriptor_phase virtual method of the init object of Is1NRelationship_<Company,Employee, CompanyEmploysEmployees> will allocate a relationship descriptor object to represent the relationship CompanyEmploysEmployees. When this phase is finished, all descriptor objects will be created.

During the link_schema_phase, the descriptor objects are connected to represent the relationships among them. For example, when the link_schema_phase virtual method of the init object of Is1N Relationship_<Company,Employee,CompanyEmploysEmployees> is called in this phase, it will link the entity descriptor objects for Company and Employee to the relationship descriptor object for CompanyEmploysEmployees. Also, when the link_schema_phase virtual method of the init object of SubtypeEntity_<Manager,Employee> is called, it will link the entity descriptor object for Manager and that for Employee to represent the subtype relationship between the two.

[5] Because Init_::initialize() will call some virtual methods in the translation units that define the static init objects, the objects are guaranteed by the language to be properly initialized.

3.2 Overriding Pure Virtuals

Template base delegation can be used to supply similar concrete implementations for pure virtual functions declared in an abstract base class. Suppose there is a pure virtual function `entity_descriptor()`, declared in the abstract base class `Entity`, which is to be overridden by each entity class to return its own descriptor object.

Using the conventional way, each entity class will have to provide an overriding virtual function. This will likely involve:

1. Declaring the virtual function `entity_descriptor()` in the definition of the entity class.
2. Declaring the descriptor object, or a pointer to it, as a static data member.
3. Defining the virtual function `entity_descriptor()`, which returns the static data member.

Alternatively, an entity class can delegate this responsibility to a template base, either `RootEntity_<...>` or `SubtypeEntity_<...>`. For example, instead of the class `Employee` overriding the pure virtual, its template base, `RootEntity_<Employee>`, can supply a concrete implementation. Similarly, `SubtypeEntity_<Manager,Employee>` can do the same for `Manager`. The template base will ensure that an appropriate override is provided for the virtual function.

Using the conventional way, the number of source lines required to override the virtual is small, but they must be repeated for each entity class. In comparison, using template base delegation, there is a constant number of lines for the base template definition, but the number of additional source lines for each addition entity class is equal to 1, that is, the one used to declare the template base.

The real advantage, however, is that the implementation of the overriding virtuals is centralized at the base template definition. If we need to change the implementation, say from declaring the descriptor object as a static data member to declaring a pointer to it, there is only one place to change in the source.

It may be argued that using macros can achieve the same results. However, using macros generally means losing the ability to debug into the macro expansions.

In comparison, the definitions of the entity classes using template base delegation are more readable, because more code is moved into the template bases.

3.3 Uniform Access

Building a "uniform-access" interface using these template bases and descriptor objects is possible. Uniform-access means essentially a general accessor method, `get_related_entities(...)`, which takes a relationship descriptor and returns a list of related entities with the specified relationship. A uniform access interface is especially useful for writing schema-independent client code, for example, a GUI (Graphical User Interface).

4 Template Base Delegation as a Design Tradeoff

Template base delegation can be viewed as a way of writing library classes. Because these library templates need to have the ability to customize themselves, they usually require more experience to write and may be more difficult to maintain. In a way, we are making the life of the client class writer easier by making the life of the base template writer more miserable. Whether this tradeoff makes sense depends on the number of client classes and client class writers versus the number of base templates and base template writers, as well as the relative complexity of the classes.

5 Conclusion

Some of the advantages of template base delegation are as follows:

1. The definition of the derived class can become more declarative.
2. The interface and data members of the base class can be customized for the derived class.
3. The derived class can specify to the template base whom to inherit from, thus preserving the application's inheritance hierarchy.
4. Runtime efficiency can be enhanced without compromising on code sharing or strong typing.
5. The derived class can inherit from two or more instantiations of the same base class template.

The technique was illustrated using the Fan templates as an alternative to the traditional `List` container templates. A practical application of the technique in the area of object modeling was also demonstrated.

6 Acknowledgements

Dave Penny and Sam Wong have provided many good insights and suggestions for this paper.

An anonymous referee suggested the use of `((void)sizeof(T))` to ensure that `class T` is defined before any downcast to `T*`. This will catch a common programming error which occurs when an upcast or downcast requires a non-zero pointer adjustment. If the definition of the derived class has not been seen by the compiler, the compiler will erroneously treat the cast as to an unrelated type without generating the required pointer adjustment code.

References

[1] Rebecca Wirfs-Brock and Brian Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. *Proceedings OOPSLA*, SIGPLAN Notices, Vol 24, #10, October 1989.

[2] Bjarne Stroustrup. *The C++ Programming Language: Second Edition.* Addison-Wesley, Reading, Massachusetts, 1991.

[3] P. Chen. "The Entity-Relation Model - Toward a Unified View of Data" ACM Transaction on Database Systems, March 1976.

[4] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual* Addison-Wesley, Reading, Massachusetts, 1990.

[5] James Coplien. *Advanced C++ Programming Styles and Idioms* Addison-Wesley, Reading, Massachusetts, 1992.

7 Appendix

Source for the Fan templates (some details omitted):

```
// If a use of the following macro causes the compiler to
// fail, the definition of class T has not been seen but is
// required.  Try rearrange your source code so that the
// definition of class T comes before the use of this macro.
//
#define ASSERT_CLASS_IS_DEFINED(T)      ((void)sizeof(T))

// If a use of the following macro fails to compile, then
// H is not inheriting from FanHead_<H,T,R>.
#define ASSERT_HEAD_INHERITS_FROM_FANHEAD(H,T,R)        \
        { FanHead_<H,T,R>*    dummy = (H*)0; }
...
```

```
class FanHeadBase_ {
  public:
    add_tip(FanTipBase_& tip);
    unsigned num_tips() const;
    ...
  protected:
    FanHeadBase_() : first_tip_(0) {}
    FanTipBase_*                        first_tip_;
};

class FanTipBase_ {
  protected:
    FanTipBase_() : head_(0), next_tip_(0) {}
    FanHeadBase_*                       head_;
    FanTipBase_*                        next_tip_;
};

template<class Head, class Tip, class Relationship>
class FanTip_;          // Forward declaration

template<class Head, class Tip, class Relationship>
class FanHead_ : protected FanHeadBase_ {
    friend class FanTip_<Head,Tip,Relationship>;
  public:
    FanHeadBase_::num_tips;
    add_tip(Tip& tip);
    ...
};

template<class Head, class Tip, class Relationship>
class FanTip_ : protected FanTipBase_ {
    friend class FanHead_<Head,Tip,Relationship>;
  public:
    Head* head() const {
        ASSERT_CLASS_IS_DEFINED(Head);
        ASSERT_HEAD_INHERITS_FROM_FANHEAD(Head,Tip,Relationship);
        return (Head*)(FanHead_<Head,Tip,Relationship>*) head_;
    }
    ...
};
```

C++ Design and Implementation Challenges in Technology Computer Aided Design Frameworks

Goodwin R. Chin (gchin@watson.ibm.com),† Dharini Sitaraman,‡ Chung Yang,‡ and Martin D. Giles (madagil@dip.eecs.umich.edu)‡

†*IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.*

‡*Solid State Electronics Laboratory, University of Michigan, Ann Arbor, MI 48109.*

1 Introduction

Interoperability of physical simulation tools is often cumbersome due to differences in domain representation. The Technology Computer Aided Design (TCAD) community [1] has realized this problem and is working on providing an object-oriented interface for representing and manipulating the state of a semiconductor wafer during process and device simulation, the Semiconductor Wafer Representation (SWR) [1, 2, 3]. We describe an implementation of components of a 3D SWR designed to support the needs of many TCAD simulators. A set of classes provide support for solid modeling operations, mesh refinement operations, and data interchange. A layered memory subsystem allows either uniprocess or client/server operation. Run-time type identification provides dynamic extensibility of data types.

The TCAD world is generally divided into process modeling (simulating the fabrication of semiconductor device structures) and device modeling (simulating the electrical characteristics of these devices). Furthermore, process modeling tools can be partitioned into topography modeling tools which change the physical shape of the device and bulk processing tools which change the material and electrical properties of the device. Examples of topography processes include lithography (pattern transfer of the integrated circuit onto the semiconductor), deposition of electrically insulating and conducting materials, and etching (removal) of these materials. Bulk processing steps include the implantation of chemical species into the bulk substrate material and the diffusion of these species in the substrate to alter electrical characteristics. Steps that involve both topography and bulk processing include thermal oxidation and silicidation — steps common in the fabrication of modern day devices.

Frameworks facilitate the development and use of modeling tools by providing infrastructure and services [4]. Many of these framework services, such as user interface management and intertool communication, are domain-independent in that their value to a framework is independent of the particular application (e.g. electrical CAD or mechanical CAD). Services such as design data representation are domain-dependent in that they are strongly coupled to the application area to be supported. Development of framework standards is being promoted by the CAD Framework Initiative (CFI), a consortium of end users and vendors worldwide, dedicated to defining interface standards for the infrastructure of design automation tools and for design data used by these tools. When vendors adhere to these standards, interoperability between a heterogeneous set of design automation tools is ensured.

The design data representation requirements for TCAD can be supported by using two representation components for wafer state — a geometry to represent the shape of the wafer and a field to represent the values of a particular solution variable at various points in the domain. Figure 1 shows a 2-D cross section of a wafer with the geometry and field portions denoted. Field values are usually associated with a mesh across the domain that is used to solve partial differential equations that describe the physical processes that are simulated.

[1] TCAD is a subset of physical simulation that primarily focuses on the simulation of the fabrication and electrical testing of semiconductors.

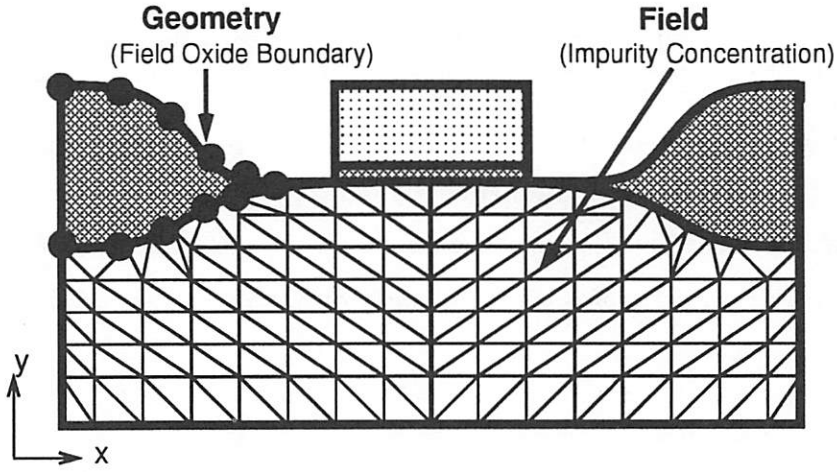


Figure 1: SWR Geometry and Fields

Section 2 describes an implementation strategy to support both geometry and mesh views of the domain. These representations pose different storage/functionality requirements and a need exists to allow easy access to both representations for the same domain. Implementation of classes to support meshing algorithms is discussed with analysis of the benefits of using C++ features such as templates and multiple inheritance with virtual base classes.

Section 3 describes implementation of a new flexible scheme to support field values on generic meshes with multiple interpolation functions. In particular, we present a run-time type library with automatic updating of virtual table pointers to allow sharing of C++ objects across multiple executables. This is an extension of the run-time typing necessary to express user-defined fields.

2 Cells

A *Cell* is a geometric object that can be used to describe the shape of the simulation domain (Figure 1). Cells can be simple (e.g. rectangle) or complex (multi-faceted polyhedral). A *Cell Complex* is formed from a set of cells with a common property (e.g. same material). *Cell Complexes* facilitate implementation of efficient material boundary queries.

Topography-based simulators typically operate in the following fashion:

- extract exposed top surface of the wafer (set of edges for a 2D wafer and a set of faces for a 3D wafer).
- using models, possibly physics-based, determine the new position of the surface. This may require information from the volumes below the surface.
- update the original wafer with the new surface. This may require adding new cells or modifying the boundaries of existing cells as volumes may be modified. Field values that are topography-related may also need to be updated.

Use of solid modeling boolean operations such as subtract and glue (Figure 2) provide robust update of the original wafer. To support efficient boolean operations, rich (in memory usage and functionality) data structures such as the "staredge" [5] are required. Fortunately many domains can be described using relatively few cells.

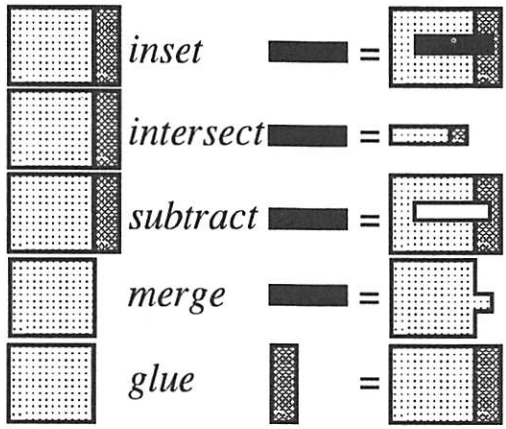


Figure 2: Set Operations

Data Structure	Memory Usage per Tetrahedral
Staredge	1.1 kBytes
Meshing Cells	450 Bytes
Minimal mesh description	76 Bytes

Table 1: Memory usage for a variety of data structures on a typical 3D mesh (729 vertices, 4184 edges, 6528 faces, 3072 tetrahedron)

Field-based simulators often use meshes to describe solution values on the domain and to discretize partial differential equations used to describe physical models (Figure 1). Cells can also be used to describe the elements within a mesh. As the number of elements in a mesh is typically orders of magnitude greater than the number of cells necessary to describe the shape of the domain, a compact data structure is required to minimize storage requirements. The minimal mesh representation adequate for efficient element-based matrix-assembly, a common operation for simulators, is an enumeration of all of the vertices (nodes) in the mesh, a list of elements, where each element contains an ordered list of nodes that define its boundary, and a list of neighboring elements. Table 1 shows the amount of storage necessary to store a tetrahedral mesh using a variety of data structures. Notice the dramatic increase in storage necessary to provide efficient boolean operations using the "staredge" data structure. Even supporting higher-level functionality such as meshing results in a 6-fold increase of storage over a minimalistic representation.

Many meshed-based simulation tools may require another service, mesh refinement, as numerical considerations may require the refinement of an element into a set of smaller elements. To support both mesh refinement and to improve the computational efficiency of matrix assembly, a *MeshingCell* class is introduced in Section 2.1.

Both the geometry cell and meshing cell classes are inherited from an interface base class as shown in Figure 3. EXPRESS-G notation[6] is used to describe relationships between classes[2]. Classes are enclosed in rectangles, with lines indicating inheritance. Relationships are unidirectional with the circle indicating the "to" side of the relationship. Errors in converting between the different representations can be minimized by having both types of cells obtain their vertices from a common pool. This ensures that corresponding vertices in the geometry and mesh representations correspond to exactly the same object rather than using numerical comparison to ensure that the vertices share the same location in space. Operations common to the base class include connec-

[2]In this paper only inheritance relationships will be shown. Containment relationships can be shown by using thin lines between classes.

tivity information such as location of the cells in space. These operations are sufficient to develop file-based translators to existing TCAD simulators. In addition, a minimal mesh description can be generated for minimal persistent storage.

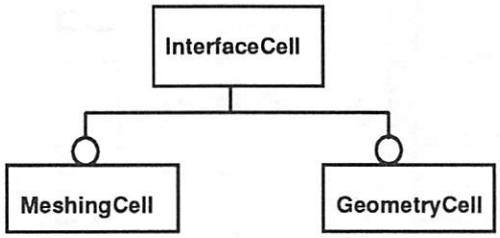


Figure 3: Cell Class Diagram in EXPRESS-G Notation

2.1 MeshingCells

To support the integration of existing meshers and to facilitate the implementation of new meshing algorithms, a set of classes have been developed as shown in Figure 4. A *MeshingCell* contains information to describe its boundary, cells of higher dimension that contain it, and the *Cell Complex* that contains it [3]. For example, edge "B" in Figure 5 would contain references to vertices "C" and "D" that describe its boundary, a reference to face "A" as a parent cell that contains it, and a reference to a cell complex. *MeshingCells* have two template arguments. The first argument is the topological dimension of the cell (e.g. a vertex has topological dimension 0, an edge has topological dimension 1, a face has topological dimension 2) and the second argument describes the dimension of the space the cell is embedded in. For example, an edge in a volume (3-space) would be denoted as *MeshingCell<1,3>*. A *MeshingCell* class might be declared as follows.

```
template <int topologicalDim, int spatialDim>
class MeshingCell
{
  private:
    Set<MeshingCell<topologicalDim-1, spatialDim> > boundary;
    Set<MeshingCell<topologicalDim+1, spatialDim> > parents;

  public:
    MeshingCell(Set<MeshingCell<topologicalDim-1, spatialDim> >&);
    Iterator<MeshingCell<topologicalDim-1, spatialDim> > boundary() const;
};
```

A *ShadowMeshingCell* is similar to a *MeshingCell* but adds additional references to interior subcells to support intermediate steps associated with mesh refinement. Consider a simple face refinement algorithm (Figure 6):

- add vertex in the center of the face (step 1).
- attach edges from the corners to the center point (steps 2 - 5).

The addition of the vertex forms an invalid cell as the point is neither on the inner or outer boundary. However use of the *ShadowMeshingCell* allows the temporary association of the point with the current face. Addition of the first edge again results in an invalid cell [4] — using the *ShadowMeshingCell* the edge can be associated with the face. Addition of the second edge causes the creation of two valid cells.

[3] This data structure is very similar to the "staredge" without explicit enumeration of "neighborhood" information (e.g. given a vertex in a particular Cell Complex, enumerate the faces which contain the vertex).

[4] we do not allow manifold cells in our representation.

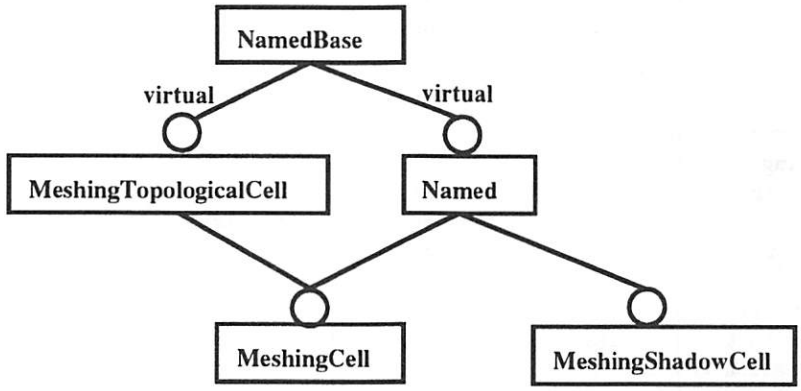


Figure 4: Meshing Classes in EXPRESS-G Notation

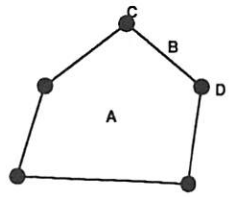


Figure 5: Cell Composition

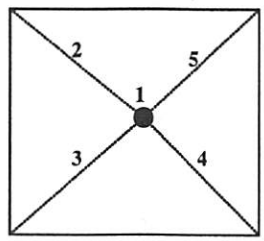


Figure 6: Refining a face

Many meshing algorithms are independent of spatial dimension (i.e. an algorithm for refining a plane should be the same whether the plane is embedded in 2-D or 3-D). However implementation of 2-D cell meshers usually assume that the cell is also embedded in 2-D — and the mesher will not be usable to mesh the surface of a volume. One way to provide surface meshing capability using an existing 2-D mesher is the following.

```
find a transformation that maps the face into the x-y plane
create a MeshingShadowCell<2,2> for the face
  map edges of the face to MeshingShadowCell<1,2> objects
  map vertices of the face to MeshingShadowCell<0,2> objects.
     this will require using the transformation on Point<3>
     objects.
  based on connectivity of the face, clone the connectivity
     hierarchy to the MeshingShadowCells.
extract information from the MeshingShadowCell<2,2> and
  send it to the existing mesher.
```

The *MeshingTopologicalCell* base class provides connectivity information in a spatially independent fashion such as a list of edges that comprise the boundary of a face. This information can be used to clone the structure of the face in the above algorithm.

2.2 Judicious Use of Virtual Base Classes

As discussed in Section 2.1, *MeshingShadowCells* can be used as reduced spatial dimension representations of *MeshingCells*. This relationship is maintained using a name matching scheme. During construction of the *MeshingShadowCell*, name matching between *MeshingTopologicalCells* and *MeshingShadowCells* of lower dimension is necessary as shown in the algorithm for spatial dimensional reduction of an edge of a face:

```
for (each vertex in the shadowed face)
  if (name of vertex == name of origin of edge)
    origin of shadowed edge =  current vertex of face
  else if (name of vertex == name of destination of edge)
    destination of shadowed edge =  current vertex of face
```

Two implementations for naming were investigated and class diagrams are shown in Figures 4 and 7 :

- Using virtual base classes to provide automatic name comparison. Classes *MeshingTopologicalCell*, *MeshingCell*, and *MeshingShadowCell* are all derived from the base class *NamedBase*.
- Explicitly providing comparison operators between *MeshingTopologicalCell* and *MeshingShadowCell* that work on name comparisons. This scheme does not require the use of virtual base classes.

The benefits of the virtual base approach are as follows:

- Virtual base classes make it clear that comparison between *MeshingTopologicalCell* and *MeshingShadowCell* objects is implemented as comparison between objects of the *Name* class and not due to some other criterion (e.g. same address).
- Virtual base classes reduce the amount of coding necessary to add another class which would need to use name comparison with either the *MeshingShadowCell* or *MeshingTopologicalCell* classes.

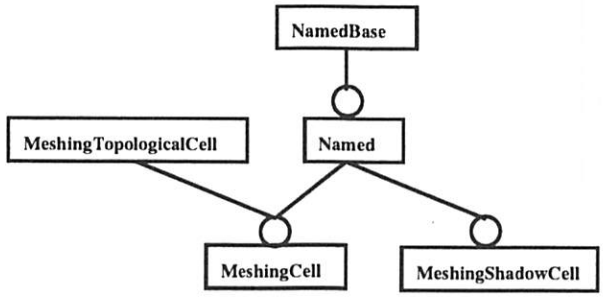


Figure 7: Class Diagram Using No Virtual Base Classes

On the other hand, there is usually a significant overhead necessary to support virtual bases because class-dependent offsets must be computed [5]. This problem is discussed in more detail by Nackman and Barton[7]. As cell shadowing occurs infrequently compared to other more time consuming operations that actually refine the cells, use of virtual base classes does not pose a large penalty in overall performance.

2.3 Use of Templates

Sometimes classes that seem perfect for templatization from a conceptual point of view become less attractive during implementation. Consider a point in space. Points could be templatized by spatial dimension and would possess common methods such as components of their coordinates and distances from other points of the same spatial dimension. A templatized class declaration might look like the following:

```
template <int spatialDim>
class Point  {
  public:
    Point(double* coordinateArray);
    double distance(const Point<spatialDim>&) const;
    double[spatialDim] components() const;
  private:
    double components[spatialDim];
};
```

While extendible from 1D to 3D, the form of the constructor and components method are awkward as the user would be forced to place the coordinates into an array rather than using a more friendly constructor such as `Point<2>::Point(double x, double y)`. As a friendly programming interface is more important than the number lines of code manually typed, the Point classes were implemented using template specialization (e.g. `Point<1>`, `Point<2>`, and `Point<3>` were defined and declared separately).

Another possible approach is to implement the *Point* class using inheritance. A base class containing the distance and components methods could be used with the corresponding derived class responsible for construction and data storage. This approach was rejected due efficiency and storage considerations associated with the use of the virtual table. Use of a virtual pointer requires the use of an extra word of storage for a *Point* — which can be a large percentage of the total space allocated for each point (2 words are necessary to store a 2D point and 3 words are necessary to store a 3D point). For compilers that cannot inline virtual functions, the overhead of calling non-inline versions of virtual methods of simple methods such as components() can be tremendous (100% time overhead on a RS/6000).

[5] Of course, the amount of overhead is implementation dependent.

The *MeshingCell* classes exhibit similar behavior for constructors. For example, a vertex is defined by a point and similarly an edge (*MeshingCell<1,3>*) is defined by a set of vertices. In addition only certain operations apply to particular objects. For example the distance operation only makes sense between two vertices. Users also complained about remembering that *MeshingCell<1,3>* was a edge in 3D. Using typedefs to translate the topological template argument into a more recognizable name help alleviate the above problem. Hence, conceptual templatization can break down in implementation.

3 Fields

A field is a mapping from a domain into a range set. For example, a topographic map displays a field which has as its domain a particular area of the country and as its range set real numbers corresponding to elevation above sea level. In the TCAD world, the field domain is usually the physical space occupied by the semiconductor wafer. The field range set depends on the physical quantity involved, and could be a real number representing atomic concentration, a real vector representing current flow, a complex number representing refractive index, or some larger combination of real and integer components. Most simulation programs use discrete approximations for fields, subdividing the domain into small, regularly shaped elements which comprise a mesh, storing range values at a limited number of locations within the mesh, and using an interpolation rule to estimate the field value at other locations across the domain. The goal of the Fields portion of SWR is to provide objects and methods to efficiently represent and manipulate mesh-based field descriptions for arbitrary range sets.

3.1 Representing Fields

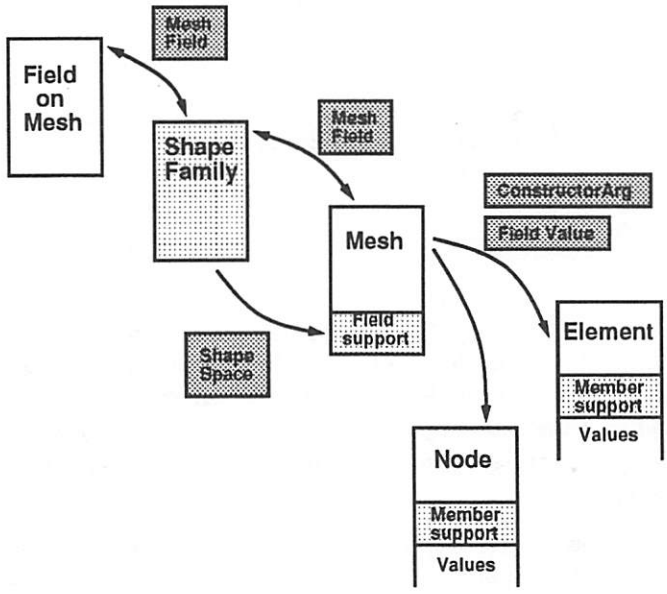


Figure 8: Objects and messages for field representation

An overview of the main field object and message classes is given in Fig. 8, where the arrows indicate messages passing between objects in our system. Field representation is divided into four levels to allow maximum sharing of information between related fields. At the bottom level are individual mesh components such as nodes and elements. A collection of mesh components filling the domain comprise a mesh. All of these objects are parameterized by spatial dimension only using the

template mechanism. The Shape Family level adds information about the storage and interpolation scheme to be used to represent the field values across the domain. This information is specific to the mesh spatial dimension but independent of the value type. At the top level, `FieldOnMesh` adds the range value type information to complete the field description. The components of a `FieldOnMesh` are connected by reference rather than by inheritance so that fields may share common information. For example, a single mesh can be used with different Shape Family interpolation rules, and a single Shape Family with different range value types. This sharing is essential for efficient use of memory since, in practice, most fields across a particular domain will use the same underlying mesh. Messaging classes are used to pass requests and responses between the levels of the field object hierarchy.

There is both good news and bad news in matching the requirements for fields against the capabilities offered by C++. On the positive side, template classes make it relatively simple to allow the user to work with fields of arbitrary range type. Our implementation of `FieldOnMesh` is parameterized by a range class and contains no type-specific information. Particular Shape Families require support for some algebraic operations from the range class so that they can perform interpolations, but if no interpolation is required then the range class has few restrictions. On the negative side, the strong type checking provided by C++ must be sidestepped in two ways. First, range class values are stored with the components of the underlying mesh. At that level there is no value type information available and the mesh components must be prepared to accept whatever mixture of value classes is supplied by the fields that reference them. In practice this does not pose a problem because the user always manipulates field value information through the `FieldOnMesh` class so value type information can always be recovered. Second, the collection of all fields across a domain form a heterogeneous set. Domain field queries can return a `FieldOnMesh` base class, but runtime type identification (RTTI) is necessary to recover the range-specific type. The set of possible range types is determined by the client applications and must be dynamically extensible so new range types can be added without rebuilding the Framework code. Our solution has been to develop a generic RTTI library supporting the operations required for Fields. The design is complicated by the distributed nature of the Framework, which requires the sharing of RTTI information between client processes.

3.2 Runtime Type Identification

Mechanisms for RTTI have been implemented in several C++ libraries (for example [8]), proposed as an extension to the C++ language[9], and finally adopted as an official language extension[10]. The RTTI scheme used here was designed based on the ideas presented by Stroustrup[11] before the language extension proposal was finalized. Our goal was to develop a separate RTTI library which could be applied to several application class hierarchies, which minimized the intrusion into the application code, and which could operate with other Framework components for storage management. A secondary goal was the ability to migrate to native language RTTI support when this eventually becomes available. Our primary application-level functions are packaged as macros to allow this without requiring application source code changes. In our Framework application, RTTI is used for all of the field classes parameterized by value type.

3.2.1 RTTI Interface

RTTI adds two practical capabilities to objects: the ability to downcast from a base class to a derived class in a type-checked way, and the ability to query the type of an object given a base class pointer or reference. Suppose we have a tree of application classes to which we wish to add this capability, such as is illustrated in Fig. 9. Adding our RTTI library requires two changes to the class header file and one change to the class implementation file. First, the class tree must contain the class `SwrTypeBase` as a base class:

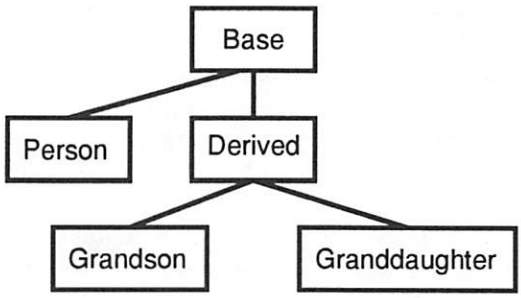


Figure 9: A simple application class hierarchy

```
class Base : public SwrTypeBase
   {
public:
    .......
   };
```

Second, any descendant of `Base` that you wish to cast into must register its type. It does this by calling a macro in the public portion of its class member declaration, passing its own class name as the argument.

```
class Derived : public Base
   {
public:
   TYPEDERIVEDH(Derived)
   ..........
   };
```

If you wish to be able to cast into the middle of a class hierarchy, each derived class must also specify its parent typed class in the hierarchy. It does this using a second form of the registration macro.

```
class Grandson : public Derived
   {
public:
   TYPEDERIVED2H(Grandson, Derived)
   ......
   };
```

Finally, the class implementation file must contain the corresponding implementation macros

```
TYPEDERIVEDC(Derived)
TYPEDERIVED2C(Grandson, Derived)
```

Although macros conveniently hide the details of the implementation, the current language specification does not allow them to work smoothly with template class names. If the derived class is a template class with multiple arguments, the comma in the class name confuses the preprocessor because it is taken as an argument separator. The RTTI library defines a preprocessor symbol `COMMA` to work around this. Template arguments are required to appear in two different forms (with and without the argument-declaration portion) so both must be supplied to the macro. The corresponding header and implementation file fragments are:

```
template <class D, class E> class Person : public Base
   {
public:
   TYPEDERIVEDH(Person)
   ........
   };

TYPEDERIVEDCT(Person, class D COMMA class E, D COMMA E)
```

At runtime, pointers and references to base classes can be cast into a derived class. Assuming br is a Base reference to a Grandson class object, we can recover the Grandson class object with

```
Grandson& gsr = REFERENCE_CAST(Grandson, br);
```

or get a pointer with

```
Grandson* gsp = POINTER_CAST(Grandson, &br);
```

We can also convert to an intermediate class in the hierarchy:

```
Derived* dp = POINTER_CAST(Derived, &br);
```

All of these calls generate a runtime error if the conversion is invalid (such as when br refers to a Granddaughter object). Alternatively, the call

```
Grandson* dp = POINTER_CAST_NOCHECK(Grandson, &br);
```

will return a null pointer is the cast is invalid. Const versions of each of these macros are provided.

The casting macros provided by our RTTI library provide an equivalent of the capabilities in the C++ language extension[10, 12]. For example, the reference casting macro could be defined using the extension as

```
#define REFERENCE_CAST(a,b) dynamic_cast<a&>(b)
```

and similarly for each of the other macros.

The second capability provided by RTTI is to allow queries of the type of an object given a base class pointer or reference. The primary purpose here is to allow comparison against a known type id. We provide this functionality using type comparison macros, such as

```
if (POINTER_TYPE_MATCH(Grandson, &br))
  ....

if (POINTER_TYPE_MATCH(Derived, &br))
  ....

if (REFERENCE_TYPE_MATCH(Grandson, br))
  ....
```

In each case, the macro returns `true` only if a conversion is possible to the specified type. In our example above, each of the tests returns `true` because br is a Base reference to a Grandson object. This behavior does not match the type comparison scheme adopted in the language extension, which would return `false` for the both of the following tests

```
if (typeid(Derived*) == typeid(&br) ||
    typeid(Derived)  == typeid(br) )
  ....
```

but true for both of

```
if (typeid(Base*)    == typeid(&br) &&
    typeid(Grandson) == typeid(br) )
  ....
```

The language extension compares strictly on the basis of the value of the supplied expression and requires an exact match rather than a possible conversion path. The extension also distinguishes between polymorphic and non-polymorphic types and defines typeid information even for built-in types, which is beyond the capabilities of our library. Nevertheless, the limited form of type matching we provide has so far been sufficient for our application.

3.2.2 RTTI Implementation

Our implementation of RTTI uses static objects to allocate and administer type information, taking care that initialization occurs exactly once for each RTTI class. The `SwrType` class holds the type information for a particular class:

```
template <class T> class SwrType
   {
private:
   static SwrTypeCode t;
public:
   static const SwrTypeCode& code() { return t; }
   static int match(const SwrTypeCode t2) { return t.match(t2); }
   };
```

The constructor for `SwrTypeCode` ensures that each typecode has a unique value. Since `SwrTypeCode` is a static member of `SwrType`, exactly one unique typecode is allocated for each object type.

The `SwrTypeBase` base class adds two virtual functions to the application hierarchy which are then overridden by the `TYPEDERIVED` macros for each derived class that is participating in the typing scheme. These functions use the static type member functions to answer queries about the current object type:

```
class SwrTypeBase
   {
public:
   virtual void* swr_type_match(const SwrTypeCode& ) const;
   virtual const SwrTypeCode& swr_type_code() const;
   };

#define TYPEDERIVED2C(T,P) \
   void* T::swr_type_match(const SwrTypeCode& t) const \
      { return SwrType< T >::match(t) ? (void*)this : P::swr_type_match(t); }\
   const SwrTypeCode& T::swr_type_code() const \
      { return SwrType< T >::code(); }
```

The swr_type_match function recursively compares the requested type against the current and immediate base type until a match is found or the top of the hierarchy is reached. The result of a match is the address of the object when viewed as that particular type, or zero if no match is found. This bit pattern is cast into void* to maintain a consistent return type, from which a typed pointer can be later recovered without altering the bit pattern.

At the top level, the dynamic type casting macros use the virtual functions to determine the corresponding cast object address.

```
#define POINTERCASTNOCHECK(T,P) ((T*)((P)->swr_type_match(SwrType<T >::code())))
```

The overhead for adding our RTTI library to a set of applications classes is to force each participating class to be polymorphic. In the worst case, this may add storage for one virtual function table pointer for each level in the class hierarchy. Runtime overhead appears at the time a dynamic cast is made, adding one function call and comparison for each level examined in the class hierarchy.

3.3 Sharing Runtime Types

Memory management for the Framework is provided by a library which can be configured in two modes. In the "local process" mode, new objects are allocated on the heap as usual. In the "shared memory" mode, pools of shared memory are managed between client processes to allow sharing of objects. We do not support simultaneous access of objects by multiple clients. Instead, self-contained collections of objects describing the fields of a particular wafer state are passed from client to client.

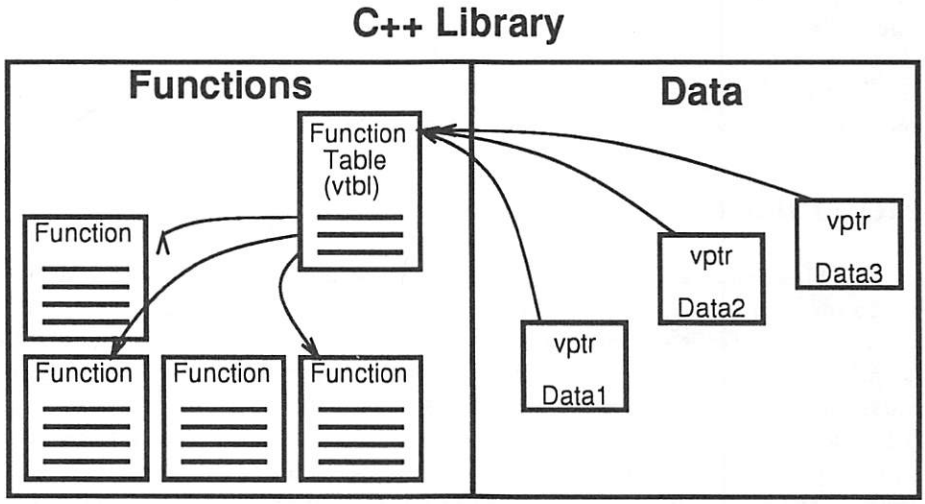


Figure 10: Components of a C++ library object

Sharing between C++ programs is considerably more complicated than with C or FORTRAN. C++ objects conceptually contain both data and functions to operate on the data. Data values can be different for each instance of the class, but the functions are common to all instances of the class. This leads to implementations where the C++ object is divided, with the data stored on the stack or heap and the functions compiled into the executable for the program, as illustrated in Fig. 10. In particular, C++ objects with virtual functions contain at least one virtual function table (vtbl) pointer which is set to the address of the vtbl in the executable which created the object. An object which is identical in all user-defined fields may be represented by a different bit

pattern if created by a different executable. This prevents naïve sharing of objects by saving and restoring dumps of memory locations or, in this case, by sharing a common memory pool between processes.

We have extended our RTTI scheme to allow objects to be "localized" to a particular executable by updating the process-specific information while leaving the other object fields untouched. This shared runtime type identification (SRTTI) allows collections of objects to be passed between clients without converting to an intermediate exchange format or otherwise packing and unpacking the object contents.

3.3.1 SRTTI Interface

Adding the capability to share objects requires two steps beyond the capability provided by the memory management library to allocate object in a shared memory area. First, the runtime type information must be extended to include additional bookkeeping. Second, clients must invoke the object localization procedure when a collection of objects moves under its control.

The change from plain RTTI to SRTTI does not require any source code changes to the application class descriptions. Turning on a preprocessor symbol `SHAREDTYPES` adds functionality to the existing `TYPEDERIVED` macros to include the infrastructure necessary to support type sharing. There are new constraints on the classes which are suitable for participation in SRTTI using the predefined macros: each base class of an SRTTI class must have a default constructor which does not modify the object state. This restriction can be removed if the user is willing to manually expand and modify the macro definition to add dummy base class construction arguments.

In our implementation, we have hidden the object localization procedure inside the mechanism that attaches a client to a pool of shared memory objects. We have assumed that each application will already have a mechanism in place to access every application-level object in the pool. At the time a new client attaches to an existing shared pool, it is required to call the virtual member function `Fixme` for every object in the pool. After this has been done, a final function call registers the current client as the object pool owner. No special action is required for subsequent object creation/deletion, nor when the client releases the memory pool.

3.3.2 SRTTI Implementation

In our plain RTTI scheme, objects are identified by the `SwrTypeCode` value returned by their `swr_type_code` member function. The type code itself is not a data member of the object – the virtual function mechanism is sufficient to ensure that the corresponding `swr_type_code` member function is called. One consequence of this is that each object type must have a corresponding unique virtual function table address because it describes a function that is unique to each typed class. SRTTI operates by building a mapping between type codes and vtbl addresses so that the localization of objects can be performed. Note that an object may contain multiple vtbl pointers. The vtbl used by the SRTTI mechanism is the one corresponding to the `SwrTypeBase` class which is a base class of all participating classes.

The SRTTI scheme operates in three phases. At static initialization time for a particular client, the SRTTI library builds a linked list of `SwrTypeCode`–vtbl pairs for all classes participating in SRTTI. This is accomplished by adding a second static member, `swrlink`, to the `SwrType` class with a constructor which accumulates the pairs. The list is held by a common base class of all `SwrType` classes called `SwrTypeShareBase`. We have also added two member functions to `SwrType`. The `Fixme` method will later be used to localize the object to a new executable. The `SwrTypeMyvtbl` is a compiler-specific routine to obtain a reference to the vtbl for this type which will be used to both read and write the vtbl information. The version shown below is appropriate for our Sun and IBM RS/6000 compilers.

```
template <class T> class SwrType : public SwrTypeShareBase
   {
private:
   static SwrTypeCode t;
   static SwrTypeLink<T> swrlink;
public:
   static SwrTypeCode& code() { return t; }
   static int match(const SwrTypeCode t2) { return t.match(t2); }
   static void Fixme(T* t,sList<SwrTypeList> *tlist);
// this is compiler specific
   static int& SwrTypeMyvtbl(SwrTypeBase& r) { return *(int*)((void*)(&r)); }
   };
```

In order for the swrlink constructor to be able to call the SwrTypeMyvtbl method, it needs an object of type T to work with. The TYPEDERIVED macros add a dummy constructor to the application class to facilitate this, but it does constrain the possible application classes to those where a temporary copy of the object can be created and deleted by the SRTTI library.

```
template <class T> SwrTypeLink<T>::SwrTypeLink()
   {
   T t(swrTypeDummyObject);
   SwrTypeShareBase::SwrTypeAddtoList(SwrType<T>::code(),SwrType<T>::SwrTypeMyvtbl(t));
   }
```

The second phase of SRTTI occurs when a new client attaches to an existing shared memory pool. This pool contains the collection of application objects and the linked list of type-vtbl pairs from the originating client. The second client also has its own linked list of type-vtbl pairs with the same type codes but different vtbl addresses. The SwrTypeBase class and TYPEDERIVED macros have been extended to include a new virtual function swr_vtbl_fix which is able to localize all of the vtbl pointers for an object using the dummy constructor and placement new syntax:

```
#define TYPEDERIVED2C(T,P) \
     .... \
   void T::swr_vtbl_fix() \
      { (void) new ((void*)this) T(swrTypeDummyObject); }
```

Initially, the second client is unable to call this virtual function because it too relies on a virtual function table pointer which still refers to the original client. When the application calls the Fixme method for a particular object, the (old) vtbl address from the object is used as a key to obtain the type code from the first client pair list. This type code is then used as a key to obtain the new vtbl address from the second client pairlist. After this new vtbl has been written into the object, the swr_vtbl_fix virtual function can be called to complete the localization process.

```
template <class T>
void SwrType<T>::Fixme(T* t,sList<SwrTypeList> *tlist)
   {
   int i = getTypeCode(t,tlist);  // map from old vtbl to typecode
   int vptr = getVtblPtr(i);      // map from typecode to new vtbl
   SwrTypeMyvtbl(*t) = vptr;      // bootstrap: first fix one vtbl entry
   t->swr_vtbl_fix();             // use virtual function to fix the rest
   }
```

After every object in the shared memory pool has been updated, the second client type-vtbl list replaces the original copy in shared memory so that it is available for use by a subsequent client.

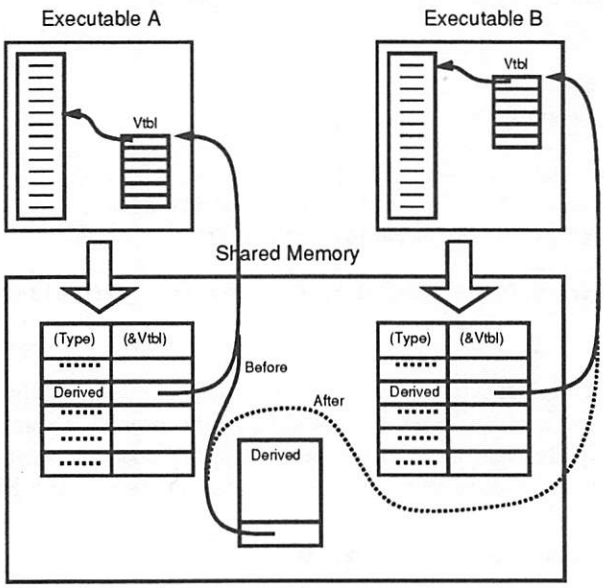


Figure 11: Sharing typed objects: before and after localization by **B**

As an example, consider the case where executable **A** has created an instance of a `Derived` object in shared memory which is now required by executable **B**, as shown in Fig. 11. Shared memory also contains the A-type-vtbl pair list for executable **A** created by the SRTTI scheme. At startup, executable **B** builds its own B-type-vtbl pair list during static initialization with same type information but its own vtbl addresses. When control of the `Derived` object passes from **A** to **B**, executable **B** calls the `Fixme` method in `Derived`, supplying the SRTTI list from executable **A**. `Fixme` uses the old vtlb information from the `Derived` object to look up its type code in the A-type-vtbl list. It then uses the type code to look up the local vtbl address in the B-type-vtbl list and updates the value stored in the `Derived` object. At this point it is free to call any virtual function from the `SwrTypeBase` class. A call to `swr_vtbl_fix` converts any remaining vtbl pointers to their appropriate local values.

SRTTI has an overhead in terms of both space and time. Space must be allocated in shared memory to hold the list of type-vtbl pairs which will have a size proportional to the number of classes participating in the SRTTI scheme. Runtime is consumed during the startup of each client as the local type-vtbl list is constructed, again proportional to the number of participating classes. There is also a runtime cost whenever a client attaches to a shared memory object pool, proportional to the number of objects in the pool. These runtime costs are small compared to alternatives requiring the full object contents to be manipulated.

4 Conclusions

We have successfully implemented components of a 3D TCAD Framework using C++. Language features within C++ were essential for achieving our design goals for development of classes representing meshes, geometries, and fields. Fields require run-time type identification and classes that operate independent of range value type to allow mesh and shape information to be shared among many fields. This scheme can be extended to support the use of shared objects in a client/server architecture by providing the means to update virtual function tables when a new client comes into scope. Cells are used for both mesh and geometry representation and are reconciled through a base class that provides containment and simple neighborhood information suitable for interchange with existing simulators. To support spatially independent mesh refinement, a shadowing scheme

is introduced which requires the use of name comparison. Two different implementations of name comparison trade off flexibility for performance. Some abstractions that lead naturally to template use in theory end up requiring template specialization to make the programming interface more user-friendly. The use of virtual functions for classes with a small number of member data and requiring many instances is very memory inefficient.

While the use of C++ provided an extremely attractive mechanism for problem abstraction along with easily maintainable and debuggable code, some real-world problems still exist with current commercial compilers. Implementation bugs related to templates and specialization are common. Other compilers take an extremely long time to link an executable containing many template classes because multiple instantiations of identical templates are formed during the template instantiation phase. These practical roadblocks are a significant factor weighing against C++ compared to more mature languages, but are not sufficient to outweigh the benefits for our particular application.

References

[1] SWR Working Group of the CFI/TCAD TSC. *Semiconductor Wafer Representation Architecture, Version 1.0.* CAD Framework Initiative, Austin, Texas, July 1992.

[2] SWR Working Group of the CFI/TCAD TSC, editor. *Semiconductor Wafer Representation Procedural Interface, Version 1.0.* CAD Framework Initiative, Austin, Texas, July 1992.

[3] M. D. Giles, D. S. Boning, G. R. Chin, W. C. Dietrich, M. S. Karasick, M. E. Law, P. K. Mozumder, L. R. Nackman, V. T. Rajan, D. M. H. Walker, R. H. Wang, and A. S. Wong. Semiconductor wafer representation for TCAD. *IEEE Trans. Computer-Aided Design*, January 1994.

[4] D. S. Harrison, A. R. Newton, R. L. Spickelmier, and T. J. Barnes. Electronic CAD Frameworks. *IEEE Proceedings*, 78(2):393–417, February 1990.

[5] M. Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, McGill University, Montreal, Quebec, 1988. (available as Cornell Univ. Dept. of Computer Science 89-976, Ithaca, NY).

[6] P. Spiby and D. A. Schenck. Express language reference manual. ISO TC 184/SC4/WG5/P3, 1991.

[7] Lee Nackman and John Barton. Base class composition with multiple derivation and virtual bases. In *Proceesings 1994 C++ USENIX Conference*, April 1994.

[8] K. E. Gorlen, S. M. Orlwo, and P. S. Plexico. *Data Abstraction and Object-Oriented Programming in C++.* Wiley, 1990.

[9] B. Stroustrup and D. Lenkov. Run-time type identification for C++ (revised). In *Proceedings of the Usenix C++ Conference*, August 1992.

[10] B. Stroustrup and D. Lenkov. Run-time type identification for C++ (revised yet again). X3J16/92-0121=WG21/N0198, 1992.

[11] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[12] Josée Lajoie. The new language extensions. *C++ Report*, 5(6):47–52, July 1993.

ASX: An Object-Oriented Framework for Developing Distributed Applications

Douglas C. Schmidt
schmidt@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine, CA 92717, (714) 856-4105

Abstract

The ADAPTIVE Service eXecutive (ASX) is a highly modular and extensible object-oriented framework that simplifies the development and configuration of distributed applications on shared memory multiprocessor platforms. This paper describes the structure and functionality of the ASX framework's object-oriented architecture. In addition, the paper presents the results of performance experiments conducted using ASX-based implementations of connection-oriented and connectionless protocols from the TCP/IP protocol family. These experiments measure the performance impact of alternative methods for parallelizing communication protocol stacks. Throughout the paper, examples are presented to indicate how the use of object-oriented techniques facilitate application extensibility, component reuse, and performance enhancement.

1 Introduction

Distributed computing is a promising technology for improving collaboration through connectivity and interworking; performance through parallel processing; reliability and availability through replication; scalability, extensibility, and portability through modularity; and cost effectiveness through resource sharing and open systems. Despite these benefits, distributed applications (such as on-line transaction processing systems, global mobile communication systems, distributed object managers, video-on-demand servers, and communication subsystem protocol stacks) are often significantly more complex to develop and configure than non-distributed applications.

A significant portion of this complexity arises from limitations with conventional tools and techniques used to develop distributed application software. Conventional application development environments (such as UNIX, Windows NT, and OS/2) lack type-safe, portable, re-entrant, and extensible system call interfaces and component libraries. For instance, endpoints of communication in the widely used socket network programming interface are identified via weakly-typed I/O descriptors that increase the potential for subtle run-time errors [1]. Another major source of complexity arises from the widespread use of development techniques based upon algorithmic decomposition [2], which limit the extensibility, reusability, and portability of distributed applications.

Object-oriented techniques offer a variety of principles, methods, and tools that help to alleviate much of the complexity associated with developing distributed applications. To illustrate how these techniques are being successfully applied in several research and commercial settings, this paper describes the structure and functionality of the ADAPTIVE Service eXecutive (ASX). ASX is an object-oriented framework containing automated tools and reusable components that collaborate to simplify the development, configuration, and reconfiguration of distributed applications on shared memory multi-processor platforms.

Components in the ASX framework are designed to decouple (1) application-independent components provided by the framework that handle interprocess communication, event demultiplexing, explicit dynamic linking, concurrency, and service configuration from (2) application-specific components inherited or instantiated from the framework that perform the services in a particular distributed application. The primary unit of configuration in the ASX framework is the *service*. A service is a portion of a distributed application that offers a single processing capability to communicating entities. Services may be simple (such as returning

the current time-of-day) or highly complex (such as a real-time distributed PBX event traffic monitor [3]). By employing object-oriented techniques to decouple the application-specific service functionality from the reusable application-independent framework mechanisms, ASX facilitates the development of applications that are significantly more extensible and portable than those based on conventional algorithmic decomposition techniques. For example, it is possible to dynamic reconfigure one or more services in an ASX-based application without requiring the modification, recompilation, relinking, or restarting of a running system [4].

In addition to describing the object-oriented architecture of the ASX framework, this paper examines results obtained by using the framework to conduct experiments on protocol stack performance in multiprocessor-based communication subsystems. In the experiments, the ASX components help control for several relevant confounding factors (such as protocol functionality, concurrency control schemes, and application traffic characteristics) in order to precisely measure the performance impact of different methods for parallelizing communication protocol stacks. For example, in the experiments described in Section 3, connectionless and connection-oriented protocol stacks were developed by specializing existing components in the ASX framework via techniques involving inheritance and parameterized types. These techniques hold the protocol functionality constant while allowing the parallel processing structure of the protocol stacks to be altered systematically in a controlled manner.

This paper is organized as follows: Section 2 outlines the primary features of the ASX framework and describes its object-oriented architecture, Section 3 examines empirical results from experiments conducted using the framework to parallelize communication protocol stacks; and Section 4 presents concluding remarks.

2 The ADAPTIVE Service eXecutive Framework

2.1 Overview

The ADAPTIVE Server eXecutive (ASX) is an object-oriented framework that is specifically targeted for the domain of distributed applications. The framework simplifies the construction of distributed applications by improving the modularity, extensibility, reusability, and portability of both the application-specific network services and the application-independent OS interprocess communication (IPC), demultiplexing, explicit dynamic linking, and concurrency mechanisms that these services utilize.

A framework is an integrated collection of components that collaborate to produce a reusable architecture for a family of applications [5]. Object-oriented frameworks are becoming increasingly popular as a means to simplify and automate the development and configuration process associated with complex application domains such as graphical user interfaces [6], databases [7], operating system kernels [8], and communication subsystems [9]. The components in a framework typically include *classes* (such as message managers, timer-based event managers, demultiplexers [10], and assorted protocol functions and mechanisms [11]), *class hierarchies* (such as an inheritance lattice of mechanisms for local and remote interprocess communication [1]), *class categories* (such as event demultiplexers [12, 13]), and *objects* (such as a service dispatch table). By emphasizing the integration and collaboration of application-specific and application-independent components, frameworks enable larger-scale reuse of software compared with simply reusing individual classes or stand-alone functions.

The ASX framework incorporates concepts from several other modular communication frameworks including System V STREAMS [14], the *x*-kernel [15], and the Conduit [9] (a survey of these and other communication frameworks appears in [16]). These frameworks all contain features that support the flexible configuration of communication subsystems by inter-connecting building-block protocol and service components. In general, these frameworks encourage the development of standard reusable communication-related components by decoupling application-specific processing functionality from the surrounding framework infrastructure. As described below, the ASX framework also contains additional features that help to further decouple application-specific service functionality from (1) the type of locking mechanisms used to synchronize access to shared objects, (2) the use of message-based vs. task-based parallel processing techniques, and (3) the use of kernel-level vs. user-level execution agents.

2.2 The Object-Oriented Architecture of ASX

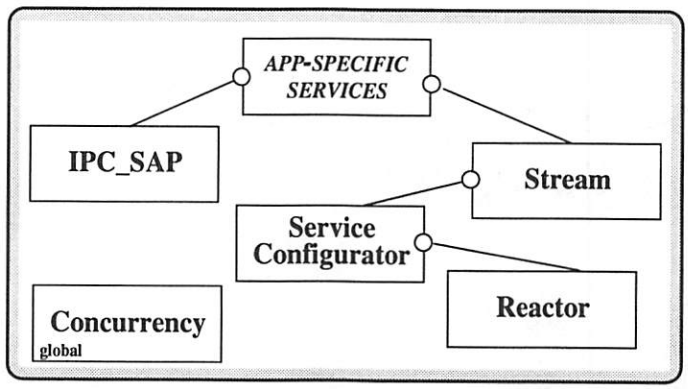


Figure 1: Class Categories in the ASX Framework

The architecture of the ASX framework was developed incrementally by generalizing from extensive design and implementation experience with a range of distributed applications including on-line transaction processing systems [13], real-time PBX performance monitoring systems [3], and multi-processor-based communication subsystems [17]. After building several prototypes and iterating through a number of alternative designs, the class categories illustrated in Figure 1 were identified and implemented. A class category is a collection of components that collaborate to provide a set of related services [2] such as communication subsystem services used to implement protocol stacks. A complete distributed application may be formed by combining components in each of the following class categories via C++ language features such as inheritance, aggregation, and template instantiation:

- `Stream Class Category` – These components are responsible for coordinating the *configuration* and run-time *execution* of a Stream, which is an object containing a set of hierarchically-related services (such as the layers in a communication protocol stack) defined by an application
- `Reactor Class Category` – These components are responsible for *demultiplexing* temporal events generated by a timer-driven callout queue, I/O events received on communication ports, and signal-based events and *dispatching* the appropriate pre-registered handler(s) to process these events
- `Service Configurator Class Category` – These components are responsible for *dynamically linking* or *dynamically unlinking* services into or out of the address space of an application at run-time
- `Concurrency Class Category` – These components are responsible for *spawning, executing, synchronizing*, and *gracefully terminating* services at run-time via one or more threads of control within one or more processes
- `IPC_SAP Class Category` – These components encapsulate standard OS local and remote IPC mechanisms (such as sockets and TLI) within a more type-safe and portable object-oriented interface

Lines connecting the class categories in Figure 1 indicate dependency relationships. For example, components that implement the application-specific services in a particular distributed application depend on the `Stream` components, which in turn depend on the `Service Configurator` components. Since components in the `Concurrency` class category are used throughout the application-specific and application-independent portions of the ASX framework they are marked with the **global** adornment. Note that the "namespaces" feature accepted recently by the ANSI C++ committee provides explicit C++ language support for these types of class category relationships.

This section examines the main components in each class category. Relationships between components in the ASX framework are illustrated throughout the paper via Booch notation [2]. Solid rectangles indicate class categories, which combine a number of related classes into a common name space. Solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate some type of link exists between two objects. Dashed clouds indicate classes; directed edges indicate inheritance relationships between classes; and an undirected edge with a small circle at one end indicates either a composition or uses relation between two classes.

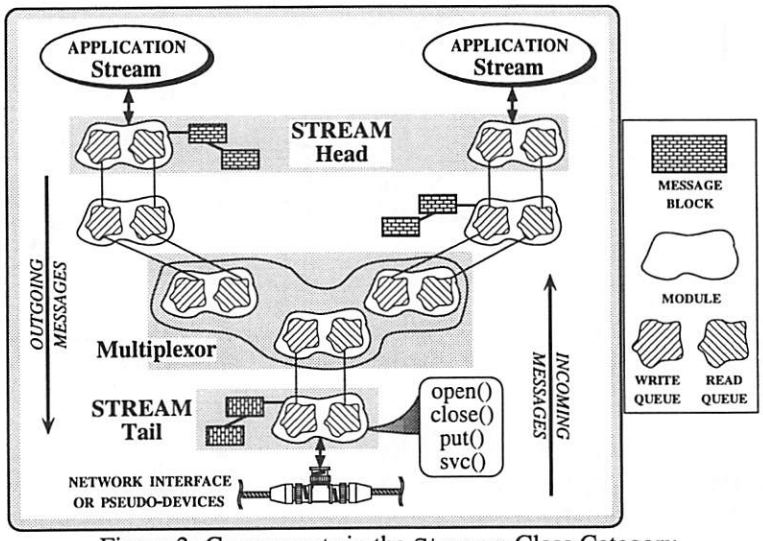


Figure 2: Components in the `Stream` Class Category

2.3 The Stream Class Category

Components in the `Stream` class category are responsible for coordinating one or more *Streams*. A Stream is an object used to configure and execute application-specific services into the `ASX` framework. As illustrated in Figure 2, a Stream contains a series of inter-connected `Modules` that may be linked together by developers at installation-time or by applications at run-time. `Modules` are objects that developers use to decompose the architecture of a distributed application into a series of inter-connected, functionally distinct layers. Each layer implements a cluster of related service-specific functions (such as an end-to-end transport service, a presentation layer formatting service, or a real-time PBX signal routing service). Every `Module` contains a pair of `Queue` objects that partition a layer into its constituent read-side and write-side service-specific processing functionality.

Any layer that performs multiplexing and demultiplexing of message objects between one or more related Streams may be developed using a `Multiplexor` object. A `Multiplexor` is a container class that provides mechanisms to route messages between one or more `Modules` in a collection of related Streams. A complete Stream is represented as an inter-connected series of independent `Module` and/or `Multiplexor` objects that communicate by exchanging messages with adjacent objects. `Modules` and `Multiplexors` may be joined together in essentially arbitrary configurations in order to satisfy application requirements and enhance component reuse.

The `ASX` framework uses C++ language features such as inheritance and parameterized types to enable developers to incorporate service-specific functionality into a Stream without requiring the modification of the basic framework components. For example, incorporating a new service layer into a Stream involves (1) inheriting from the `Queue` interface and selectively overriding several member functions (described below) in the subclass to implement service-specific functionality, (2) allocating a new `Module` that contains two instances (one for the read-side and one for the write-side) of the service-specific `Queue` subclass, and (3) inserting the `Module` into a Stream object. Service-specific functions in adjacent inter-connected `Queues` collaborate by exchanging typed messages via a uniform message passing interface.

To avoid reinventing familiar terminology, many C++ class names in the `Stream` class category correspond to similar componentry available in the System V STREAMS framework. However, the techniques used to support extensibility and concurrency in the two frameworks are significantly different. For example, adding service-specific functionality to the `ASX Stream` classes is performed by inheriting from several interfaces and implementations defined by existing framework components. Using inheritance to add service-specific functionality provides greater type-safety than the pointer-to-function idiom used in System V STREAMS. As described in Section 2.6.1 below, the `ASX Stream` classes also completely redesign and reim-

plement the co-routine-based, "weightless"[1] service processing mechanisms used in System V STREAMS. These ASX changes enable more effective use of multiple PEs on shared memory multi-processing platforms by reducing the likelyhood of deadlock and simplifying flow control between Queues in a Stream.

The remainder of this section discusses the primary components of the ASX Stream class category (*i.e.*, Stream class, the Module class, the Queue class, and the Multiplexor class) in detail.

2.3.1 The STREAM Class

The STREAM class defines the application interface to a Stream. A STREAM object contains a stack of one or more hierarchically-related services that provide applications with a bi-directional get/put-style interface for sending and receiving data and control messages to the service-specific Module layers within a particular Stream. The STREAM class also implements mechanisms that allow applications to configure a Stream at run-time by inserting and removing objects of the Module class that is described next.

2.3.2 The Module Class

A Module object is used to attach a layer of service-specific functionality together with the other Module objects that are connected together to form a Stream. By default, two standard Module objects (Stream_Head and Stream_Tail) are installed automatically when a Stream is opened. These standard Modules interpret pre-defined framework control messages that may be passed through a Stream at run-time. For incoming data, the Stream_Tail class typically transforms network packets received by network interfaces or pseudo-devices into a canonical internal message format recognized by other components in a Stream. Likewise, for outgoing data it transforms messages from their internal format into network packets. The Stream_Head class provides a message buffering interface between an application and a Stream. I/O between an application and a Stream occurs synchronously when the Stream_Head Module appears at the top of a Stream. However, if the Stream_Head is omitted, messages percolating up a Stream are delivered into the address space of an application asynchronously.

2.3.3 The Queue Abstract Class

Each Module object contains a pair of pointers to objects that are service-specific subclasses of the Queue abstract class.[2] One Queue subclass handles read-side processing for messages sent upstream to its Module layer and the other handles write-side processing messages send downstream to its Module layer. The Queue class is an abstract class since its interface defines four pure virtual member functions: open, close, put, and svc. Defining Queue as an abstract class enhances reuse by decoupling the general-purpose components provided by the Stream class category from the service-specific subclasses that inherit from and use these components. Likewise, the use of pure virtual member functions allows the C++ compiler to ensure that a subclass of Queue honors its obligation to provide the following service-specific functionality:

• **Initialization and Termination Member Functions:** Subclasses derived from Queue must implement open and close member functions that perform service-specific Queue initialization and termination activities. These activities typically allocate and free resources such as connection control blocks, I/O descriptors, and synchronization locks. The open and close member functions of a Module's write-side and read-side Queue subclasses are invoked automatically by the ASX framework when the Module is inserted or removed from a Stream, respectively.

• **Service-Specific Processing Member Functions:** Subclasses of Queue also must define the put and svc member functions, which perform service-specific processing functionality on messages that arrive at a Module layer in a Stream. When messages arrive at the head or the tail of a Stream, they are escorted through a series of inter-connected Queues as a result of invoking the put and/or svc member function of

[1] A weightless process executes on a run-time stack that is also used by other processes. This greatly complicates programming and increases the potential for deadlock. For example, a weightless process may not suspend execution to wait for resources to become available or events to occur [18].

[2] An abstract class in C++ provides an interface that contains at least one *pure virtual member function* [19]. A pure virtual member function provides only an interface declaration, without any accompanying definition. Subclasses of an abstract class must provide definitions for all its pure virtual member functions before any objects of the class may be instantiated.

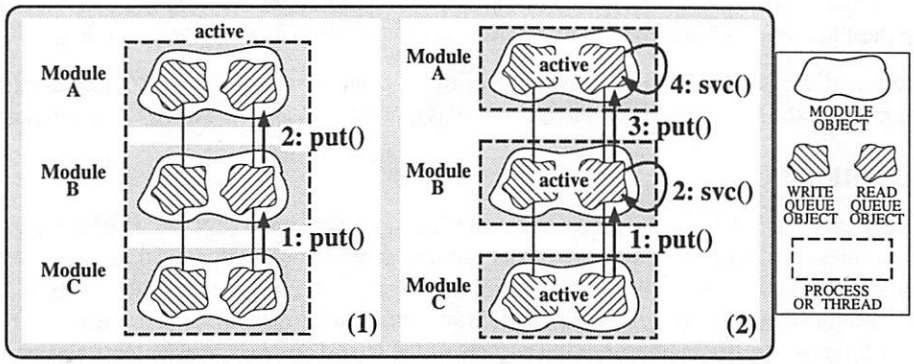


Figure 3: Alternative Methods for Invoking `put` and `svc` Member Functions

each `Queue` in the Stream.

A `put` member function is invoked when a `Queue` at one layer in a Stream passes a message to an adjacent `Queue` in another layer. The `put` member function runs *synchronously* with respect to its caller, *i.e.,* it borrows the thread of control from the `Queue` that originally invoked its `put` member function. This thread of control typically originates either "upstream" from an application process, "downstream" from a pool of threads that handle I/O device interrupts [15], or internal to the Stream from an event dispatching mechanism (such as a timer-driven callout queue used to trigger retransmissions in a connection-oriented transport protocol `Module`).

The `svc` member function is used to perform service-specific processing *asynchronously* with respect to other `Queues` in its Stream. Unlike `put`, the `svc` member function is not directly invoked from an adjacent `Queue`. Instead, it is invoked by a separate thread associated with its `Queue`. This thread executes the `Queue`'s `svc` member function, which runs an event loop that continuously blocks waiting for messages to arrive on the `Queue`'s `Message_List`. A `Message_List` is a standard component in a `Queue` that is used to buffer a sequence of data messages and control messages for subsequent processing in the `svc` member function. When messages arrive, the `svc` member function dequeues the messages and performs the `Queue` subclass's service-specific processing tasks.

Within a `put` or `svc` member function, a message may be forwarded to an adjacent `Queue` in the Stream by passing the message via the `put_next` utility member function. `Put_next` calls the `put` member function of the next `Queue` residing in an adjacent layer. This invocation of `put` may borrow the thread of control from the caller and handle the message immediately (*i.e.,* the synchronous processing approach illustrated in Figure 3 (1)). Conversely, the `put` member function may enqueue the message and defer handling to its `svc` member function that is executing in a separate thread of control (*i.e.,* the asynchronous processing approach illustrated in Figure 3 (2)). As discussed in Section 3, the particular processing approach that is selected often has a significant impact on performance and ease of programming.

In addition to the four pure virtual member function interfaces, each `Queue` also contains a number of reusable utility member functions (such as `put_next`, `getq`, and `putq`) that may be used by service-specific subclasses to query and/or modify the internal state of a `Queue` object. This internal state includes a pointer to the adjacent `Queue` on a Stream, a back-pointer to a `Queue`'s enclosing `Module` (which enables it to locate its sibling), a `Message_List`, and a pair of high and low water mark variables that are used to implement layer-to-layer flow control between adjacent `Modules` in a Stream. The high water mark indicates the amount of bytes of messages the `Message_List` is willing to buffer before it becomes flow controlled. The low water mark indicates the level at which a previously flow controlled `Queue` is no longer considered to be flow controlled.

Two types of messages may appear on a `Message_List`: simple and composite. A simple message contains a single `Message_Block` and a composite message contains multiple `Message_Blocks` linked together. Composite messages generally consist of a *control* block followed by one or more *data* blocks. A control block contains bookkeeping information (such as destination addresses and length fields), whereas

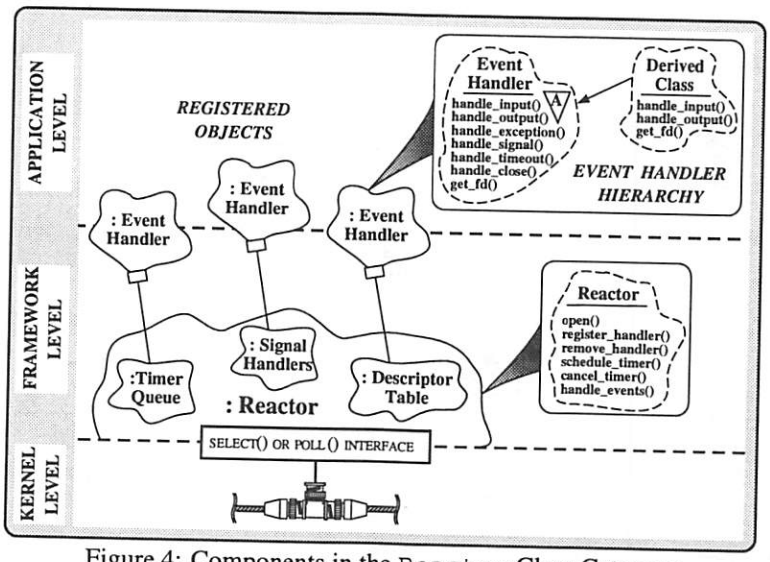


Figure 4: Components in the `Reactor` Class Category

data blocks contain the actual contents of a message. The overhead of passing `Message_Blocks` between `Queues` is minimized by passing pointers to messages rather than copying data.

2.3.4 The Multiplexor Class

A `Multiplexor` is a container class that provides mechanisms to demultiplex messages between one or more `Modules` in a collection of inter-related Streams. `Multiplexors` are typically used to route `Message_Blocks` between inter-related streams (such as those used to implement complex protocol families in the Internet and the ISO OSI reference models). A `Multiplexor` is implemented as a C++ template class parameterized by an *external identifier* (such as a network address, port number, or type-of-service field) and an *internal identifier* (such as a pointer to a `Module`). These template parameters are instantiated by service-specific Stream components to produce specialized `Multiplexor` objects that perform efficient intra-Stream message routing. Each `Multiplexor` object contains a set of `Modules` that may be linked above and below the `Multiplexor` in essentially arbitrary configurations.

2.4 The Reactor Class Category

Components in the `Reactor` class category are responsible for demultiplexing (1) temporal events generated by a timer-driven callout queue, (2) I/O events received on communication ports, and (3) signal events and dispatching the appropriate pre-registered handler(s) to process these events. The `Reactor` encapsulates the functionality of the `select` and `poll` I/O demultiplexing mechanisms within a portable and extensible C++ wrapper [12, 13]. `Select` and `poll` are UNIX system calls that detect the occurrence of different types of input and output events on one or more I/O descriptors simultaneously. To improve portability, the `Reactor` provides the same interface regardless of whether `select` or `poll` is used as the underlying I/O demultiplexor. In addition, the `Reactor` contains mutual exclusion mechanisms designed to perform callback-style programming correctly and efficiently in a multi-threaded event processing environment.

The `Reactor` contains a set of member functions illustrated in Figure 4. These member functions provide a uniform interface to manage objects that implement various types of service-specific handlers. Certain member functions register, dispatch, and remove I/O descriptor-based and signal-based handler objects from the `Reactor`. Other member functions schedule, cancel, and dispatch timer-based handler objects. As shown in Figure 4, these handler objects all derive from the `Event_Handler` abstract base class. This class specifies an interface for event registration and service handler dispatching.

The `Reactor` uses the virtual member functions in the `Event_Handler` interface to integrate the demultiplexing of I/O descriptor-based and signal-based events together with timer-based events. I/O descriptor-based events are dispatched via the `handle_input`, `handle_output`, `handle_exceptions`, and

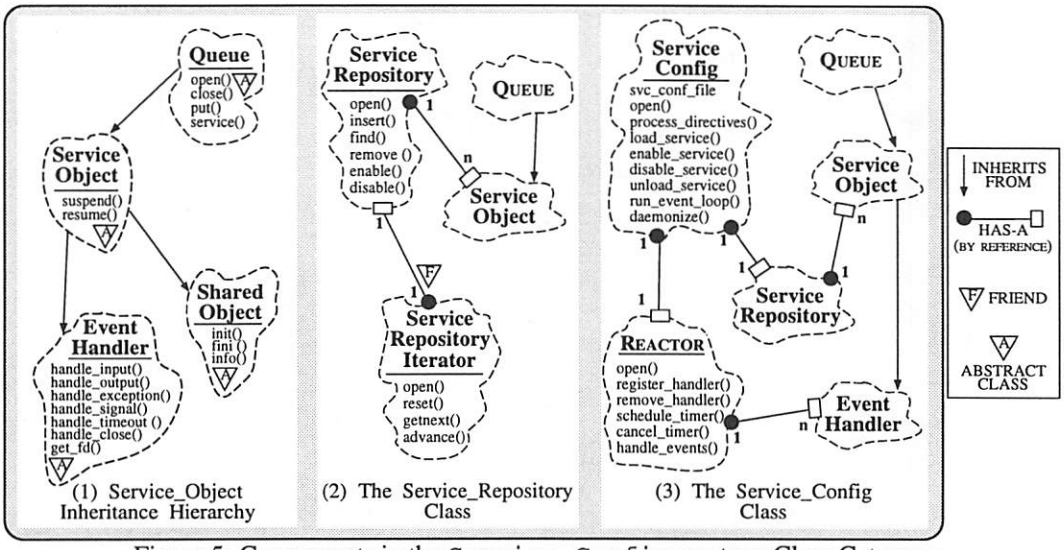


Figure 5: Components in the `Service Configurator` Class Category

`handle_signal` member functions. Timer-based events are dispatched via the `handle_timeout` member function. Subclasses of `Event_Handler` may augment the base class interface by defining additional member functions and data members. In addition, virtual member functions in the `Event_Handler` interface may be selectively overridden to implement application-specific functionality. Once the pure virtual member functions in the `Event_Handler` base class have been supplied by a subclass, an application may define an instance of the resulting composite service handler object.

When an application instantiates and registers a composite I/O descriptor-based service handler object, the `Reactor` extracts the underlying I/O descriptor from the object. This descriptor is stored in a table along with I/O descriptors from other registered objects. Subsequently, when the application invokes its main event loop, these descriptors are passed as arguments to the underlying OS event demultiplexing system call (*e.g.*, `select` or `poll`). As events associated with a registered handler object occur at run-time, the `Reactor` automatically detects these events and dispatches the appropriate member function(s) of the service handler object associated with the event. This handler object then becomes responsible for performing its service-specific functionality before returning control to the main `Reactor` event-loop.

2.5 The Service Configurator Class Category

Components in the `Service Configurator` class category are responsible for explicitly linking or unlinking services dynamically into or out of the address space of an application at run-time. Explicit dynamic linking enables the configuration and reconfiguration of application-specific services without requiring the modification, recompilation, relinking, or restarting of an executing application [4]. The `Service Configurator` components discussed below include the the `Service_Object` inheritance hierarchy (Figure 5 (1)), the `Service_Repository` class (Figure 5 (2)), and the `Service_Config` class (Figure 5 (3)).

2.5.1 The Service_Object Inheritance Hierarchy

The `Service_Object` class is the focal point of a multi-level hierarchy of types related by inheritance. The interfaces provided by the abstract classes in this type hierarchy may be selectively implemented by service-specific subclasses in order to access `Service Configurator` features. These features provide transparent dynamic linking, service handler registration, event demultiplexing, service dispatching, and run-time control of services (such as suspending and resuming a service temporarily). By decoupling the service-specific portions of a handler object from the underlying `Service Configurator` mechanisms, the effort necessary to insert and remove services from an application at run-time is significantly reduced.

The Service_Object inheritance hierarchy consists of the Event_Handler and Shared_Object abstract base classes, as well as the Service_Object abstract derived class. The Event_Handler class was described above in the Reactor Section 2.4. The behavior of the other classes in the Service Configurator class category is outlined below:

• **The Shared_Object Abstract Base Class:** The Shared_Object base class specifies an interface for dynamically linking and unlinking objects into and out of the address space of an application. This abstract base class exports three pure virtual member functions: init, fini, and info. These functions impose a contract between the reusable components provided by the Service Configurator and service-specific objects that utilize these components. By using pure virtual member functions, the Service Configurator ensures that a service handler implementation honors its obligation to provide certain configuration-related information. This information is subsequently used by the Service Configurator to automatically link, initialize, identify, and unlink a service at run-time.

The init member function serves as the entry-point to an object during run-time initialization. This member function is responsible for performing application-specific initialization when an object derived from Shared_Object is dynamically linked. The info member function returns a humanly-readable string that concisely reports service addressing information and documents service functionality. Clients may query an application to retrieve this information and use it to contact a particular service running in the application. The fini member function is called automatically by the Service Configurator class category when an object is unlinked and removed from an application at run-time. This member function typically performs termination operations that release dynamically allocated resources (such as memory or synchronization locks).

The Shared_Object base class is defined independently from the Event_Handler class to clearly separate their two orthogonal sets of concerns. For example, certain applications (such as a compiler or text editor) might benefit from dynamic linking, though it might not require timer-based, signal-based, or I/O descriptor-based event demultiplexing. Conversely, other applications (such as an ftp server) require event demultiplexing, but might not require dynamic linking.

• **The Service_Object Abstract Derived Class:** Support for dynamic linking, event demultiplexing, and service dispatching is typically necessary to automate the dynamic configuration and reconfiguration of application-specific services in a distributed system. Therefore, the Service Configurator class category defines the Service_Object class, which is a composite class that combines the interfaces inherited from both the Event_Handler and the Shared_Object abstract base classes. During development, application-specific subclasses of Service_Object may implement the suspend and resume virtual member functions in this class. The suspend and resume member functions are invoked automatically by the Service Configurator class category in response to certain external events (such as those triggered by receipt of the UNIX SIGHUP signal). An application developer may define these member functions to perform actions necessary to suspend a service object without unlinking it completely, as well as to resume a previously suspended service object. In addition, application-specific subclasses must implement the four pure virtual member functions (init, fini, info, and get_fd) that are inherited (but not defined) by the Service_Object subclass.

To provide a consistent environment for defining, configuring, and using Streams, the Queue class in the Stream class category is derived from the Service_Object inheritance hierarchy (illustrated in Figure 5 (1)). This enables hierarchically-related, application-specific services to be linked and unlinked into and out of a Stream at run-time.

2.5.2 The Service_Repository Class

The ASX framework supports the configuration of applications that contain one or more Streams, each of which may have one or more inter-connected service-specific Modules. Therefore, to simplify run-time administration, it may be necessary to individually and/or collectively control and coordinate the Service_Objects that comprise an application's currently active services. The Service_Repository is an object manager that coordinates local and remote queries and updates involving the services offered by an application. A search structure within the object manager binds service names (represented as ASCII strings) with instances of composite Service_Objects (represented as C++ object code). A service name uniquely identifies an instance of a Service_Object stored in the repository.

Each entry in the Service_Repository contains a pointer to the Service_Object portion of an service-specific C++ derived class (shown in Figure 5 (2)). This enables the Service Configurator to automatically load, enable, suspend, resume, or unload Service_Objects from a Stream dynamically. The repository also maintains a handle to the underlying shared object file for each dynamically linked Service_Object. This handle is used to unlink and unload a Service_Object from a running application when its service is no longer required. An iterator class is also supplied along with the Service_Repository. This class may be used to visit every Service_Object in the repository without compromising data encapsulation.

2.5.3 The Service_Config Class

As illustrated in Figure 5 (3), the Service_Config class integrates several other ASX components (such as the Service_Repository and the Reactor). The resulting composite Service_Config component is used to automate the static and/or dynamic configuration of concurrent applications that contain one or more Streams. The Service_Config class uses a configuration file to guide its configuration and reconfiguration activities. Each application may be associated with a distinct configuration file. This file characterizes the essential attributes of the service(s) offered by an application. These attributes include the location of the shared object file for each dynamically linked service, as well as the parameters required to initialize a service at run-time. By consolidating service attributes and installation parameters into a single configuration file, the administration of Streams within an application is simplified. Application development is also simplified by decoupling the configuration and reconfiguration mechanisms provided by the framework from the application-specific attributes and parameters specified in a configuration file. Further information on the configuration format utilized by the Service_Config class is presented in [4].

2.6 The Concurrency Class Category

Components in the Concurrency class category are responsible for spawning, executing, synchronizing, and gracefully terminating services at run-time via one or more threads of control within one or more processes. The following section discusses the two main groups of classes (Synch and Thr_Manager) in the Concurrency class category.

2.6.1 The Synch Classes

Components in the Stream, Reactor, and Service Configurator class categories described above contain a minimal amount of internal locking mechanisms to avoid over-constraining the granularity of the synchronization strategies used by an application [18]. In particular, only components in the ASX framework that would not function correctly in a multi-threaded environment (such as enqueueing Message_Blocks onto a Message_List, demultiplexing Message_Blocks onto internal Module addresses stored in a Multiplexor object, or registering an Event_Handler object with the Reactor) are protected by synchronization mechanisms provided by the Synch classes. The Synch classes provide type-safe C++ interfaces for two basic types of synchronization mechanisms: Mutex and Condition objects [20]. A Mutex object is used to ensure the integrity of a shared resource that may be accessed concurrently by multiple threads of control. A Condition object allows one or more cooperating threads to suspend their execution until a condition expression involving shared data attains a particular state. The ASX framework also provides a collection of more sophisticated concurrency control mechanisms (such as Monitors, Readers_Writer locks, and recursive Mutex objects) that build upon the two basic synchronization mechanisms described below.

A Mutex object may be used to serialize the execution of multiple threads by defining a critical section where only one thread executes its code at a time. To enter a critical section, a thread invokes the Mutex::acquire member function. To leave a critical section, a thread invokes the Mutex::release member function. These two member functions are implemented via adaptive spin-locks that ensure mutual exclusion by using an atomic hardware instruction. An adaptive spin-lock operates by polling a designated memory location using the hardware instruction until (1) the value at this location is changed by the thread that currently owns the lock (signifying that the lock has been released and may now be acquired) or (2) the thread that is holding the lock goes to sleep (at which point the thread that is spinning also goes to sleep to avoid needless polling) [21]. On a shared memory multi-processor, the overhead incurred by a spin-lock is

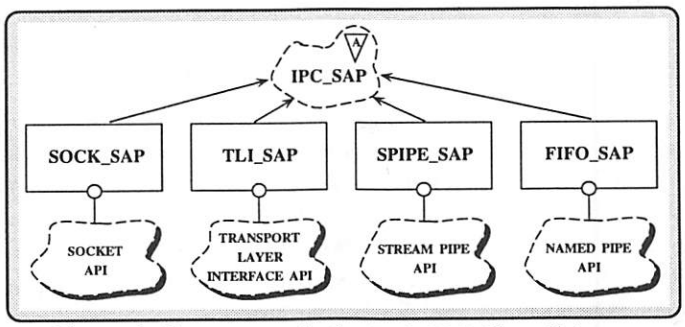


Figure 6: Components in the IPC_SAP Class Category

relatively minor since polling affects only the local instruction and data cache of the CPU where the thread is spinning. A spin-lock is a simple and efficient synchronization mechanism for certain types of short-lived resource contention. For example, in the ASX framework, each Message_List in a Queue object contains a Mutex object that prevents race conditions from occurring when Message_Blocks are enqueued and dequeued concurrently by multiple threads of control running in adjacent Queues.

A Condition object is a somewhat different synchronization mechanism that enables a thread to suspend itself indefinitely (via the Condition::wait member function) until a condition expression involving shared data attains a particular state. When another cooperating thread indicates that the state of the shared data has changed (by invoking the Condition::signal member function), the associated Condition object wakes up the suspended thread. The newly awakened thread then re-evaluates the condition expression and potentially resumes processing if the shared data is now in an appropriate state. For example, each Message_List in the ASX framework contains a pair of Condition objects (named notfull and notempty), in addition to a Mutex object. These Condition objects implement flow control between adjacent Queues. When one Queue attempts to insert a Message_Block into a neighboring Queue that has reached its high water mark, the Message_List::enqueue member function performs a wait operation on the notfull condition object. This operation atomically relinquishes the PE and puts the calling thread to sleep awaiting notification when flow control conditions abate. Subsequently, when the number of bytes in the flow controlled Queue's Message_List fall below its low water mark, the thread running the blocked Queue is automatically awakened to finish inserting the message and resume its processing tasks.

Unlike Mutex objects, Condition object synchronization is not implemented with a spin-lock since there is generally no indication of how long a thread must wait for a particular condition to be signaled. Therefore, Condition objects are implemented via sleep-locks that trigger a context switch to allow other threads to execute. Section 3 discusses the consequences of spin-locks vs. sleep-locks on application performance.

2.6.2 The Thr_Manager Class

The Thr_Manager class contains a set of mechanisms that manage groups of threads that collaborate to implement collective actions (such as a pool of threads that render different portions of a large image in parallel). The Thr_Manager class provides a number of mechanisms (such as suspend_all and resume_all) that suspend and resume a set of collaborating threads atomically. This feature is useful for distributed applications that execute one or more services concurrently. For example, when initializing a Stream composed of Modules that execute in separate threads of control and collaborate by passing messages between threads, it is important to ensure that all Queues in the Stream are completely inter-connected before allowing messages to flow through the Stream. The mechanisms in the Thr_Manager class allow these initialization activities to occur atomically.

2.7 The IPC_SAP Class Category

Components in the IPC_SAP class category encapsulate standard OS local and remote IPC mechanisms (such as sockets and TLI) within a more a type-safe and portable object-oriented interface. IPC_SAP stands

for "InterProcess Communication Service Access Point." As shown in Figure 6, a forest of class categories are rooted at the `IPC_SAP` base class. These class categories includes `SOCK_SAP` (which encapsulates the socket API), `TLI_SAP` (which encapsulates the TLI API), `SPIPE_SAP` (which encapsulates the UNIX SVR4 STREAM pipe API), and `FIFO_SAP` (which encapsulates the UNIX named pipe API).

Each class category in `IPC_SAP` is itself organized as an inheritance hierarchy where every subclass provides a well-defined subset of local or remote communication mechanisms. Together, the subclasses within a hierarchy comprise the overall functionality of a particular communication abstraction (such as the Internet-domain or UNIX-domain protocol families). Inheritance-based hierarchical decomposition facilitates the reuse of code that is common among the various `IPC_SAP` class categories. For example, the C++ interface to the lower-level UNIX OS device control system calls like `fcntl` and `ioctl` are inherited and shared by all the other components in the `IPC_SAP` class category.

3 Communication Subsystem Performance Experiments

To illustrate how the components of the `ASX` framework are used in practice, this section describes results from performance experiments that measure the impact of alternative methods for parallelizing communication subsystems. A communication subsystem is a distributed system that consists of *protocol functions* (such as routing, segmentation/reassembly, connection management, end-to-end flow control, remote context management, demultiplexing, message buffering, error protection, session control, and presentation conversions) and *operating system mechanisms* (such as process management, asynchronous event invocation, message buffering, and layer-to-layer flow control) that support the implementation and execution of protocol stacks that contain hierarchically-related protocol functions [16].

Advances in VLSI and fiber optic technology are shifting performance bottlenecks from the underlying networks to the communication subsystem [22]. Designing and implementing multi-processor-based communication subsystems that execute protocol functions and OS mechanisms in parallel is a promising technique for increasing protocol processing rates and reducing latency. To significantly increase communication subsystem performance, however, the speed-up obtained from parallel processing must outweight the context switching and synchronization overhead associated with parallel processing.

A context switch is generally triggered when an executing process either voluntarily or involuntarily relinquishes the processing element (PE) it is executing upon. Depending on the underlying OS and hardware platform, performing a context switch may require dozens to hundreds of instructions due to the flushing of register windows, instruction and data caches, instruction pipelines, and translation look-aside buffers [23]. Synchronization mechanisms are necessary to serialize access to shared objects (such as messages, message queues, protocol context records, and demultiplexing tables) related to protocol processing. Certain methods of parallelizing protocol stacks incur significant synchronization overhead from managing locks associated with processing these shared objects [24].

A number of *process architectures* have been proposed as the basis for parallelizing communication subsystems [22, 25, 24]. A process architecture binds one or more processing elements (PEs) together with the protocol tasks and messages that implement protocol stacks in a communication subsystem. As shown in Figure 7 (1), the three basic elements that form the foundation of a process architecture are (1) the *processing elements* (PEs), which are the underlying execution agents for both protocol and application code, (2) *control messages and data messages*, which are typically sent and received from one or more applications or from network devices, and (3) *protocol processing tasks*, which perform protocol-related functions upon messages as they arrive and depart from the communication subsystem.

Two fundamental types of process architectures (*task-based* and *message-based*) may be created by structuring the three basic process architecture elements shown in Figure 7 (1) in different ways. Task-based process architectures are formed by binding one or more PEs to different units of protocol functionality (shown in Figure 7 (2)). In this architecture, tasks are the active objects, whereas messages processed by the tasks are the passive objects. Parallelism is achieved by executing protocol tasks in separate PEs and passing data messages and control messages between the tasks/PEs. In contrast, message-based process architectures are formed by binding the PEs to the protocol control messages and data messages received from applications and network interfaces (as shown in Figure 7 (3)). In this architecture, messages are the active objects, whereas tasks are the passive objects. Parallelism is achieved by escorting multiple data messages

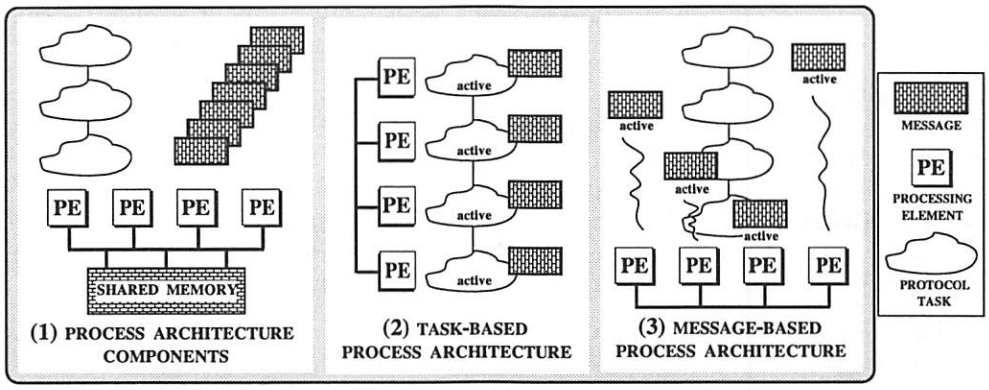


Figure 7: Process Architecture Components and Interrelationships

and control messages on separate PEs simultaneously through a stack of protocol tasks. Section 3 examines how the choice of process architecture significantly affects context switch and synchronization overhead. A survey of alternative process architectures appears in [16].

Selecting an effective process architecture is an important design decision in application domains other than communication subsystems. For example, real-time PBX monitoring systems [3] and video-on-demand servers also perform non-communication-related tasks (such as database query processing) that benefit from a carefully structured approach to parallelism. This section focuses primarily upon the impact of process architectures on communication subsystem performance since network protocol behavior and functionality is well-understood and the terminology is relatively well-defined. Moreover, a large body of literature exists with which to compare performance results presented in Section 3. The remainder of this section describes relevant aspects of performance experiments that measure the impact of different process architectures on connectionless and connection-oriented protocol stacks.

3.1 Multi-processor Platform

All experiments were conducted on an otherwise idle Sun 690MP SPARCserver, which contains 4 SPARC 40 MHz processing elements (PEs), each capable of performing at 28 MIPs. The operating system used for the experiments is release 5.3 of SunOS, which provides a multi-threaded kernel that allows multiple system calls and device interrupts to execute in parallel [21]. All the process architectures in these experiments execute protocol tasks in separate *unbound* threads multiplexed over 1, 2, 3, or 4 SunOS *lightweight processes* (LWPs) within a process. SunOS 5.3 maps each LWP directly onto a separate kernel thread. Since kernel threads are the units of PE scheduling and execution in SunOS, this mapping enables multiple LWPs (each executing protocol processing tasks in an unbound thread) to run in parallel on the SPARCserver's PEs.

Rescheduling and synchronizing a SunOS LWP involves a kernel-level context switch. The time required to perform a context switching between two LWPs was measured to be approximately 30 *u*secs. During this time, the OS performs system-related overhead (such as flushing register windows, instruction and data caches, instruction pipelines, and translation lookaside buffers) on the PE and therefore does not perform protocol processing. Measurements also revealed that it requires approximately 3 micro-seconds to acquire or release a `Mutex` object implemented with a SunOS adaptive spin-lock. Likewise, measurements indicated that approximately 90 micro-seconds are required to synchronize two LWPs using `Condition` objects implemented using SunOS sleep-locks. The larger amount of overhead for the `Condition` operations compared with the `Mutex` operations occurs from the more complex locking algorithms involved, as well as the additional context switching incurred by the SunOS sleep-locks that implement the `Condition` objects.

3.2 Communication Protocols

Two types of protocol stacks are used in the experiments, one based on the connectionless UDP transport protocol and the other based on the connection-oriented TCP transport protocol. The protocol stacks contain

the data-link, transport, and presentation layers.[3] The presentation layer is included in the experiments since it represents a major bottleneck in high-performance communication systems due primarily to the large amount of data movement overhead it incurs [26, 25].

Both the connectionless and connection-oriented protocol stacks were developed by specializing existing components in the ASX framework via techniques involving inheritance and parameterized types. These techniques are used to hold the protocol stack functionality constant while systematically varying the process architecture. For example, each protocol layer is implemented as a `Module` whose read-side and write-side inherit standard interfaces and implementations from the `Queue` class. Likewise, synchronization and demultiplexing mechanisms required by a protocol layer or protocol stack are parameterized using template arguments that are instantiated based on the type of process architecture being tested.

Data-link layer processing in each protocol stack is performed by the `DLP Module`. This `Module` transforms network packets received from a network interface or loop-back device into a canonical message format used by the Stream components. The transport layer component of the protocol stacks are based on the UDP and the TCP implementation in the BSD 4.3 Reno release. The 4.3 Reno TCP implementation contains the TCP header prediction enhancements, as well as the slow start algorithm and congestion avoidance features. The UDP and TCP transport protocols are configured into the ASX framework via the `UDP` and `TCP Modules`, respectively.

Presentation layer functionality is implemented in the `XDR Module` using marshalling routines produced by the ONC eXternal Data Representation (XDR) stub generator. The ONC XDR stub generator automatically translates a set of type specifications into marshalling routines that encode/decode implicitly-typed messages before/after they are exchanged among hosts that may possess heterogeneous processor byte-orders. The ONC presentation layer conversion mechanisms consist of a type specification language (XDR) and a set of library routines that implement the appropriate encoding and decoding rules for built-in integral types (*e.g.,* char, short, int, and long) and real types (*e.g.,* float and double). In addition, these library routines may be combined to produce marshalling routines for arbitrary user-defined composite types (such as record/structures, unions, arrays, and pointers). Messages exchanged via XDR are implicitly-typed, which improves marshalling performance at the expense of flexibility. The `XDR` functions selected for both the connectionless and connection-oriented protocol stacks convert incoming and outgoing messages into and from variable-sized arrays of structures containing both integral and real values. This conversion processing involves byte-order conversions, as well as dynamic memory allocation and deallocation.

3.3 Process Architectures

3.3.1 Design of the Task-based Process Architecture

The ASX components illustrated in Figure 8 implement a task-based process architecture for the TCP-based connection-oriented and UDP-based connectionless protocol stacks. Protocol-specific processing for the data-link and transport layer are performed in two `Modules` clustered together into one thread. Likewise, presentation layer and application interface processing is performed in two `Modules` clustered into a separate thread. These threads cooperate in a producer/consumer manner, operating in parallel on the header and data fields of multiple incoming and outgoing messages.

The `LP_DLP::svc` and `LP_XDR::svc` member functions perform service-specific processing in parallel within a Stream of `Modules`. When messages are inserted into a `Queue`'s `Message_List`, the `svc` member function dequeues the messages and performs the `Queue` subclass's service-specific processing tasks (such as data-link layer processing or presentation layer processing). Depending on the "direction" of a message (*i.e.,* incoming or outgoing), each cluster of `Modules` performs its associated protocol functions before passing the message to an adjacent `Module` running asynchronously in a separate thread. Messages are not be copied when passed between adjacent `Queues` since threads all share a common address space. However, moving messages between threads typically invalidates per-PE data caches.

[3] Preliminary tests indicated that the PE, bus, and memory performance of the SunOS multi-processor platform was capable of processing messages through the protocol stack at a much faster rate than the platform's 10 Mbps Ethernet network interface was capable of handling. Therefore, for the process architecture experiments, the network interface was simulated with a single-copy pseudo-device driver operating in loop-back mode. For this reason, the routing and segmentation/reassembly functions of the network layer processing were omitted from these experiments since both the sender and receiver portions of the test programs reside on the same host machine.

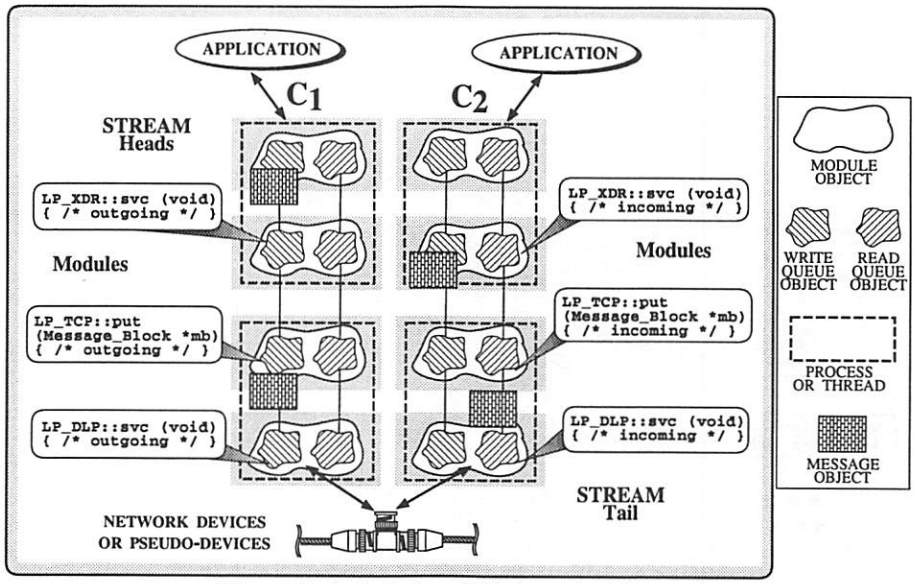


Figure 8: A Task-based Process Architecture

The connectionless and connection-oriented task-based process architecture protocol stacks are designed in a similar manner. The primary difference is that the objects in the connectionless transport layer `Module` implement the simpler UDP functionality that does not generate acknowledgements, keep track of round-trip time estimates, or manage congestion windows. The design of the task-based process architecture test driver always uses PEs in multiples of two: one for the cluster of data-link and transport layer processing `Modules` and the other for the cluster of presentation layer and application interface processing `Modules`.

3.3.2 Design of the Message-based Process Architecture

Figure 9 illustrates a message-based process architecture for the connection-oriented protocol stack. When an incoming message arrives, it is handled by the `MP_DLP::svc` member function, which manages a pool of pre-spawned threads. Each message is associated with a separate thread that escorts the message synchronously through a series of inter-connected `Queues` in a Stream by making an upcall [27] to the `put` member function in the adjacent `Queue` at each higher layer in the protocol stack. Each `put` member function executes the protocol tasks associated with that layer. The `MP_TCP::put` member function utilizes `Mutex` objects that serialize access to per-connection control blocks as separate messages from the same connection ascend the protocol stack in parallel.

The connectionless message-based protocol stack is structured in a similar manner. However, the connectionless protocol stack perform the simpler set of UDP functionality. Unlike the `MP_TCP::put` member function, the `MP_UDP::put` member function processes each message concurrently and independently, without explicitly preserving inter-message ordering. This reduces the amount of synchronization operations required to locate and update shared resources.

3.4 C++ Features Used to Simplify Process Architecture Implementation

Many of the protocol functions, process architecture synchronization mechanisms, and `ASX` framework support components (such as demultiplexing and message buffering classes) are reused throughout the process architecture test programs described above. For example, process architecture-specific synchronization strategies may be instantiated by selectively instrumenting protocol functions with different types of mutual exclusion mechanisms. When combined with C++ language features such as inheritance and parameterized types, these objects help to decouple protocol processing functionality from the concurrency control scheme used by a particular process architecture.

For example, `Multiplexor` objects use a `Map_Manager` component to demultiplex incoming messages to `Modules`. `Map_Manager` is a search structure container class that is parameterized by an external

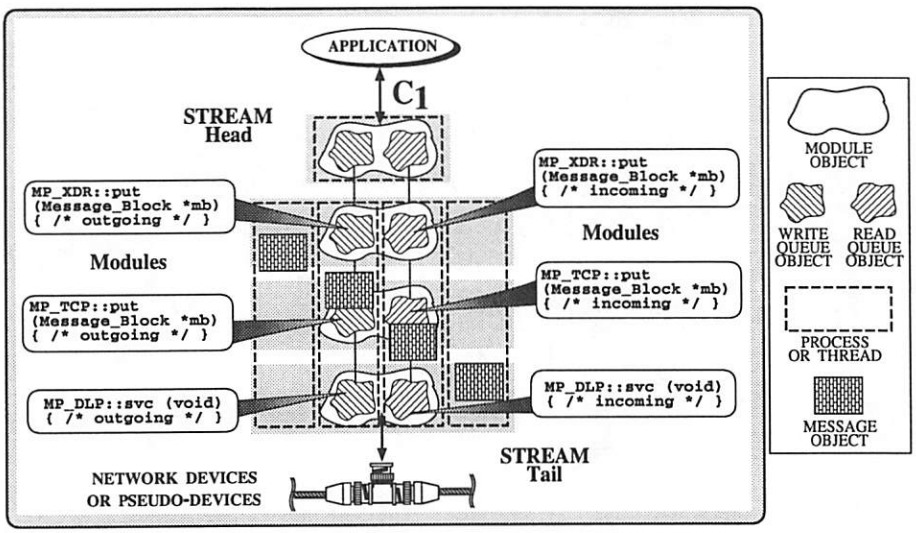


Figure 9: A Message-based Process Architecture

ID, internal ID, and a mutual exclusion mechanism, as follows:

```
template <class EXT_ID, class INT_ID, class MUTEX>
class Map_Manager {
public:
  // ...
  int find (EXT_ID ext_id, INT_ID &int_id);

private:
  MUTEX lock;
  // ...
```

The type of `MUTEX` that this template class is instantiated with depends upon the particular choice of process architecture. For instance, the `Map_Manager` used in the message-based implementation of the TCP protocol stack described in Section 3.3.2 is instantiated with the following class parameters:

```
typedef Map_Manager <TCP_Addr, TCB, Mutex> MP_Map_Manager;
```

This particular instantiation of `Map_Manager` locates the transport control block (`TCB`) associated with the `TCP` address of an incoming message. The `Map_Manager` class uses the `Mutex` class described in Section 2.6.1 to ensure that its `find` member function executes as a critical section. This prevents race conditions with other threads that are inspecting or inserting entries into the connection map in parallel.

In contrast, the task-based process architecture implementation of the TCP protocol stack described in Section 3.3.1 does not require the same type of concurrency control within a connection. In this case, demultiplexing is performed within the `svc` member function in the `LP_DLP` read `Queue` of the data-link layer `Module`, which runs in its own separate thread of control. Therefore, the `Map_Manager` used for the connection-oriented task-based process architecture is instantiated with a different `MUTEX` class, as follows:

```
typedef Map_Manager <TCP_Addr, TCB, Null_Mutex> LP_Map_Manager;
```

The implementation of the `acquire` and `release` member functions in the `Null_Mutex` class are essentially "no-op" inline functions that may be removed completely by the compiler optimizer.

The `ASX` framework employs a C++ idiom that involves using a class constructor and destructor to acquire and release locks on synchronization objects, respectively [28]. The `Mutex_Block` class illustrated below defines a "block" of code over which a `Mutex` object is acquired and then automatically released when the block of code is exited and the object goes out of scope:

```
template <class MUTEX>
```

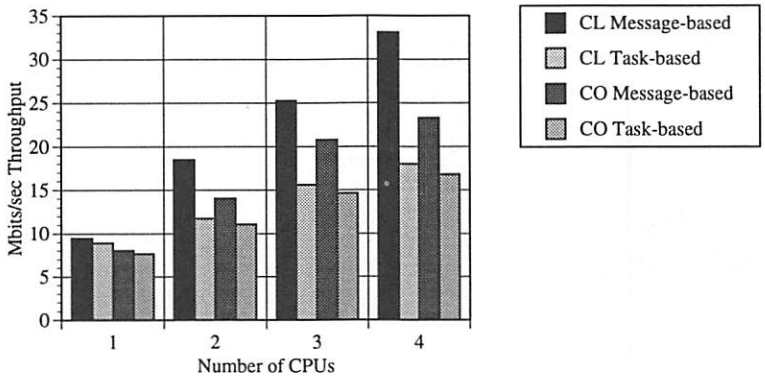


Figure 10: Process Architecture Throughput

```
class Mutex_Block
{
public:
        Mutex_Block (MUTEX &m): mutex (m) { this->mutex.acquire (); }
        ~Mutex_Block (void) { this->mutex.release (); }
private:
        MUTEX &mutex;
}
```

This idiom is used in the implementation of the `Map_Manager::find` member function, as follows:

```
template <class EXT_ID, class INT_ID, class MUTEX> int
Map_Manager<EXT_ID, INT_ID, MUTEX>::find (EXT_ID ext_id, INT_ID &int_id)
{
  Mutex_Block<MUTEX> monitor (this->lock);

  if (/* ext_id is successfully located */)
    return 0;
  else
    return -1;
}
```

When the `find` member function returns, the destructor for the `Mutex_Block` object automatically releases the `Mutex` lock. Note that the `Mutex` lock is release regardless of which arm in the `if/else` statement returns from the `find` member function. In addition, this C++ idiom also properly releases the lock if an exception is raised during processing in the body of the `find` member function.

3.5 Process Architecture Experiment Results

This section presents measurement results obtained from the data reception portion of the connection-oriented and connectionless protocol stacks implemented using the task-based and message-based process architectures described above. Three types of measurements were obtained for each combination of process architecture and protocol stack: *total throughput*, *context switching overhead*, and *synchronization overhead*.

Total throughput was measured by holding the protocol functionality, application traffic patterns, and network interfaces constant and systematically varying the process architecture to determine the resulting performance impact. Each benchmarking session consisted of transmitting 10,000 4K byte messages through an extended version of the widely available `ttcp` protocol benchmarking tool. The original `ttcp` program measures the processing resources and overall user and system time required to transfer data between a transmitter process and a receiver process communicating via TCP or UDP. The flow of data is uni-directional, with the transmitter flooding the receiver with a user-selected number of data buffers. Various sender and receiver parameters (such as the number of data buffers transmitted and the size of application messages and protocol windows) may be selected at run-time.

The version of `ttcp` used in our experiments was enhanced to allow a user-specified number of communicating applications to be measured simultaneously. This feature measured the impact of multiple con-

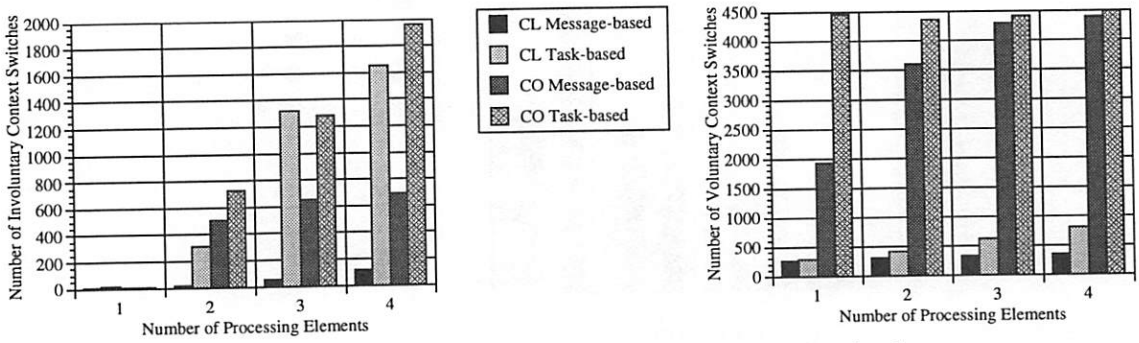


Figure 11: Process Architecture Context Switching Overhead

nections on process architecture performance (two connections were used to test the connection-oriented protocols). The `ttcp` program was also modified to use the `ASX`-based protocol stacks configured via the process architectures described in Section 3.5. To measure the impact of parallelism on throughput, each test was run using 1, 2, 3, and 4 PEs successively, using 1, 2, 3, or 4 LWPs, respectively. Furthermore, each test was performed several times to detect the amount of spurious interference incurred from other internal OS tasks (the variance between test runs proved to be insignificant).

Context switching and synchronization measurements were obtained to help explain differences in the throughput results. These metrics were obtained from the SunOS 5.3 `/proc` file system, which records the number of voluntary and involuntary context switches incurred by threads in a process, as well as the amount of time spent waiting to obtain and release locks.

Figure 10 illustrates throughput (measured in Mbits/sec) as a function of the number of PEs for the task-based and message-based process architectures used to implement the connection-oriented (CO) and connectionless (CL) protocol stacks. The results in this figure indicate that parallelization definitely improves performance. Each 4 Kbyte message effectively required an average of between 3.2 and 3.9 milliseconds to process when 1 PE was used, but only .9 to 1.9 milliseconds to process when 4 PEs were used. However, the message-based process architectures significantly outperformed their task-based counterparts as the number of PEs increased from 1 to 4. For example, the performance of the connection-oriented task-based process architecture was only slightly better using 4 PEs (approximately 16 Mbits/sec, or 1.92 milliseconds per-message processing time) than the message-based process architecture was using 2 PEs (14 Mbits/sec, or 2.3 milliseconds per-message processing time). Moreover, if a larger number of PEs had been available, it appears likely that the performance improvement gained from parallel processing in the task-based process architectures would have leveled off sooner than the message-based tests due to the higher rate of growth for context switching and synchronization shown in Figure 11 and Figure 12.

Figure 11 illustrates the number of voluntary and involuntary context switches incurred by the process architectures measured in this study. A voluntary context switch is triggered when a thread puts itself to sleep until certain resources (such as I/O devices or synchronization locks) become available. For example, when a protocol task attempts to acquire a resource that may not become available immediately (such as obtaining a message from an empty `Message_List`), the protocol task puts itself to sleep by invoking the `wait` member function of a `Condition` object. This action causes the OS kernel to preempt the current thread and perform a context switch to another thread that is capable of executing protocol tasks immediately. Figure 11 indicates the number of involuntary context switches incurred by the process architectures. An involuntary context switch occurs when the OS kernel preempts a running thread. For example, the OS preempts running threads periodically when their time-slice expires in order to schedule other threads to execute.

As shown in Figure 11, the task-based tests incur significantly more voluntary context switches than the message-based process architectures, which accounts for the substantial difference in overall throughput. The primary reason for this difference is that the locking mechanisms used for the message-based process architectures utilize adaptive spin-locks (which rarely trigger a context switch), rather than the sleep-locks used for task-based process architectures (which *do* trigger a context switch). The task-based process archi-

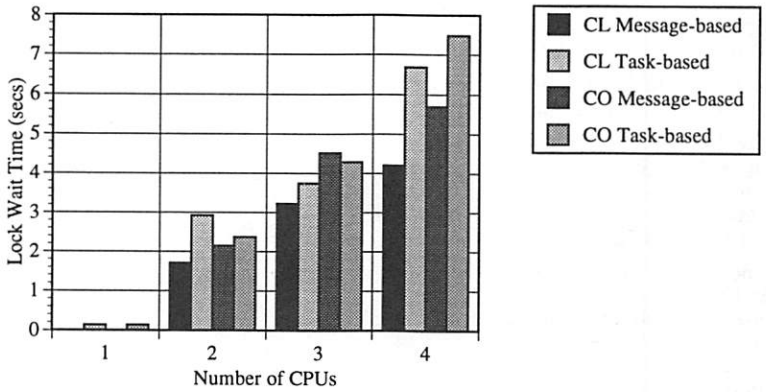


Figure 12: Process Architecture Locking Overhead

tectures also exhibited greater levels of involuntary context switching. This is due mostly to the fact that they required more time to process the 10,000 messages and were therefore pre-empted a greater number of times.

Figure 12 indicates the amount of execution time `/proc` reported as being devoted to waiting to acquire and release locks in the connectionless and connection-oriented benchmark programs. As with context switching benchmarks, the message-oriented process architectures incurred considerably less synchronization overhead, particularly when 4 PEs were used. As before, the spin-locks used by message-based process architecture reduce the amount of time spent synchronizing, in comparison with the sleep-locks used by the task-based process architectures.

4 Concluding Remarks

Despite an increase in the availability of operating system and hardware platforms that support networking and parallel processing [21, 29, 18, 30], developing distributed applications that effectively utilize parallel processing remains a complex and challenging task. The ADAPTIVE Service eXecutive (ASX) provides an extensible object-oriented framework that simplifies the development of distributed applications on shared memory multi-processor platforms. The ASX framework employs a variety of advanced OS mechanisms (such as multi-threading and explicit dynamic linking), object-oriented design techniques (such as encapsulation, hierarchical classification, and deferred composition) and C++ language features (such as parameterized types, inheritance, and dynamic binding) to enhance software quality factors (such as robustness, ease of use, portability, reusability, and extensibility) without degrading application performance. In general, the object-oriented techniques and C++ features enhance the software quality factors, whereas the advanced OS mechanisms improve application functionality and performance.

A key aspect of concurrent distributed application performance involves the type of process architecture selected to structure parallel processing of tasks in an application. Empirical benchmark results reported in this paper indicate that the task-based process architectures incur relatively high-levels of context switching and synchronization overhead, which significantly reduces their performance. Conversely, the message-based process architectures incur much less context switching and synchronization, and therefore exhibit higher performance. The ASX framework helped to contributed to these performance experiments by providing a set of object-oriented components that decouple the protocol-specific functionality from the underlying of process architecture, thereby simplifying experimentation.

The ASX framework components described in this paper are freely available via anonymous ftp from `ics.uci.edu` in the file `gnu/C++_wrappers.tar.Z`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ADAPTIVE project [31] at the University of California, Irvine. Components in the ASX framework have been ported to both UNIX and Windows NT and are currently being used in a number of commercial products including the Bellcore Q.port ATM signaling software product, the Ericsson EOS family of PBX monitoring applications, and the network management portion of the Motorola Iridium mobile communications system.

References

[1] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[2] G. Booch, *Object Oriented Analysis and Design with Applications (2^{nd} Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

[3] D. C. Schmidt and P. Stephenson, "An Object-Oriented Framework for Developing Network Server Daemons," in *Proceedings of the Second C++ World Conference*, (Dallas, Texas), SIGS, Oct. 1993.

[4] D. C. Schmidt and T. Suda, "The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons," in *The Proceedings of the Second International Workshop on Configurable Distributed Systems*, (Pittsburgh, PA), IEEE, Mar. 1994.

[5] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, pp. 22–35, June/July 1988.

[6] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *USENIX C++ Workshop November*, November 1987.

[7] D. Batory and S. W. O'Malley, "The Design and Implementation of Hierarchical Software Systems Using Reusable Components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, Oct. 1991.

[8] R. Campbell, V. Russo, and G. Johnson, "The Design of a Multiprocessor Operating System," in *USENIX C++ Conference Proceedings*, pp. 109–126, USENIX Association, November 1987.

[9] J. M. Zweig, "The Conduit: a Communication Abstraction in C++," in *USENIX C++ Conference Proceedings*, pp. 191–203, USENIX Association, April 1990.

[10] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas, "Tools for Implementing Network Protocols," *Software Practice and Experience*, vol. 19, pp. 895–916, September 1989.

[11] D. C. Schmidt, B. Stiller, T. Suda, A. Tantawy, and M. Zitterbart, "Language Support for Flexible, Application-Tailored Protocol Configuration," in *18th Conference on Local Computer Networks*, (Minneapolis, Minnesota), pp. 369–378, Sept. 1993.

[12] D. C. Schmidt, "The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2)," *C++ Report*, vol. 5, February 1993.

[13] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.

[14] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[15] N. C. Hutchinson and L. L. Peterson, "The *x*-kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[16] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[17] D. C. Schmidt and T. Suda, "Measuring the Impact of Alternative Parallel Process Architectures on Communication Subsystem Performance," in *Submitted to the 14^{th} International Conference on Distributed Computing Systems*, (Poznan, Poland), IEEE, June 1994.

[18] S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan, "Pitfalls in Multithreading SVR4 STREAMS and other Weightless Processes," in *Winter USENIX Conference*, (San Diego, CA), pp. 85–106, Jan. 1993.

[19] Bjarne Stroustrup and Margret Ellis, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[20] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[21] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[22] M. Zitterbart, B. Stiller, and A. Tantawy, "A Model for High-Performance Communication Subsystems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 507–519, May 1993.

[23] J. C. Mogul and A. Borg, "The Effects of Context Switches on Cache Performance," in *Proceedings of the 4^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Santa Clara, CA), ACM, Apr. 1991.

[24] Mats Bjorkman and Per Gunningberg, "Locking Strategies in Multiprocessor Implementations of Protocols," in *SIGCOMM Symposium on Communications Architectures and Protocols*, (San Francisco, California), ACM, 1993.

[25] M. Goldberg, G. Neufeld, and M. Ito, "A Parallel Approach to OSI Connection-Oriented Protocols," in *Proceedings of the 3^{rd} IFIP Workshop on Protocols for High-Speed Networks*, (Stockholm, Sweden), May 1992.

[26] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *SIGCOMM Symposium on Communications Architectures and Protocols*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[27] D. D. Clark, "The Structuring of Systems Using Upcalls," in *Proceedings of the Tenth Symposium on Operating Systems Principles*, (Shark Is., WA), 1985.

[28] G. Booch and M. Vilot, "Simplifying the Booch Components," *C++ Report*, vol. 5, June 1993.

[29] A. Garg, "Parallel STREAMS: a Multi-Process Implementation," in *Winter USENIX Conference*, (Washington, D.C.), Jan. 1990.

[30] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young, "Mach Threads and the Unix Kernel: The Battle for Control," in *USENIX Summer Conference*, USENIX, August 1987.

[31] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment," *Journal of Concurrency: Practice and Experience*, vol. 5, pp. 269–286, June 1993.

Interface Translation and Implementation Filtering

Mark A. Linton
Silicon Graphics
linton@sgi.com

Douglas Z. Pan
Stanford University
pan@panda.stanford.edu

Abstract

Separating interface from implementation in C++ requires a set of conventions for defining classes. Using an interface definition language, we can ensure that an interface does not contain any implementation details. To simplify the definition of separate interfaces, the translator that generates C++ class declarations should be flexible and convenient to use.

As part of the Fresco user interface system, we have developed an interface translator called Ix. In addition to generating C++ classes and stubs for distributed access, Ix can "filter" implementation code to automate as much of the code as possible. Filtering also gives the programmer more control over where and how the code is generated. We have built an initial implementation of Fresco using Ix, and our experience has been that using Ix has made programming with interfaces easier than using C++ directly.

1. Introduction

Defining an object in terms of its interface as distinct from its implementation is one of the basic tenets of object-oriented programming. C++ provides a mechanism for separating interfaces from implementation, but not a policy with which one can enforce the separation. In addition, an object interface defined in C++ might not be suitable for a remote implementation, that is, a situation where the caller of a member function resides in a different address space than the target object.

The Object Management Group (OMG) has developed the Interface Definition Language (IDL) as a standard way to specify objects that may be accessed remotely. IDL is part of OMG's Common Object Request Broker Architecture (CORBA) that defines the mechanisms for transparent access to objects in a network.

Using IDL to define object interfaces has three potential advantages over using C++:

- IDL enforces a policy of separating interfaces and implementation.
- IDL can be mapped in a natural way to more than one programming language, so an interface can be used conveniently from languages other than C++.
- IDL supports distribution—the sender and receiver of an operation may be in different address spaces and therefore on different machines in a network.

However, a potential drawback is that using IDL (and interfaces in general) might increase the burden on the programmer by requiring knowledge about the interface language, its mapping to C++, and how to use the tool that translates interfaces. All this knowledge must be acquired in addition to doing the actual implementation work.

For the Fresco[3] user interface system, we wanted to use IDL because of its potential advantages but were concerned that we and other programmers would find the burdens unacceptably high. As part of Fresco, we have therefore built a tool, called Ix, to simplify the use of interfaces.

Ix gives the programmer several options for how to translate interfaces to C++ class definitions. One such option is the choice of whether or not to use virtual base class derivation, which allows programmers to avoid the overhead and complexity of virtual base classes in some cases.

In addition to translating interface definitions to C++ class definitions and generating stubs for remote calls, Ix can perform "filtering" on an implementation header or source file to output those parts of the implementation that can be automated. One kind of filtering Ix performs is to generate the function type signature of an operation defined in the interface. Using this feature, a programmer need only enter the function signature once in the interface definition. All other instances of the signature, whether in class definitions or implementation files, can be generated automatically.

We have also built a tool, called I2mif, to simplify the documentation of a collection of interfaces. I2mif creates a document interchange file from a collection of interface files, copying specially-marked comments as well as the interface definitions.

2. Fresco

Fresco defines an object-oriented API for graphical user interfaces that covers functionality in Xlib and Xt and adds support for structured graphics and application embedding. Fresco covers a broad range of functionality—buttons, menus, rectangles, white space, text editors, and movie players can all be Fresco objects.

We decided to specify Fresco in IDL rather than C++ to obtain the advantages of using interfaces. We particularly needed the Fresco specification to be free of any implementation details. Support for multiple languages was attractive though not required. Additionally, distribution is desirable to support application embedding. As user environments become more object-oriented, turning applications into objects, users want distribution to be available for the individual components in a user interface just as is now available for individual applications using the X Window System.

For example, one might want to edit a single document containing text, a spreadsheet, and a drawing, where a separate editor is available for each type of component. The three component editors should be able to run in separate address spaces, perhaps on different machines, while the user should see them composed in a single unit.

An added benefit of using interfaces is the ability to generate run-time type information automatically without modifying the C++ compiler. CORBA defines two mechanisms that rely on run-time type information: narrowing and dynamic invocation. Narrowing effectively queries an object to see if it supports a particular interface and is similar to the C++ dynamic cast operation[7].

Dynamic invocation is a way to interpret a member function call, performing operation lookup and parameter checking at run-time rather than compile time. For Fresco, dynamic invocation is a particularly attractive way to support scripting. To investigate the viability of this approach, we implemented a script interface to Fresco called Dish (Dynamic Invocation SHell) using the Tcl language and interpreter[6]. Dish is a relatively small application (about 1,100 lines of C++) that uses dynamic invocation to evaluate commands that create and manipulate Fresco objects. All Fresco operations defined in IDL are automatically available in Dish, without any special registry or manual setup.

3. Translating interfaces

At first glance, the translation of an IDL interface to a C++ class appears straightforward:

- an IDL operation maps to C++ virtual functions
- an IDL attribute maps to a pair of C++ virtual functions

- IDL data types map either directly to C++ data types or to C++ classes that define the behavior of the IDL data type.

However, the important issue is to what extent the generated C++ class is abstract, that is, whether the generated class contains any data or non-pure member functions.

3.1 Member data

Typically, a generated class will inherit from some common base class. Proxy objects—surrogates that refer to an implementation in another address space—must store the information needed to access the remote implementation. It is tempting to put this data in the common base class object. However, generating a class that has data, either directly or inherited from a common base class, has two significant drawbacks.

The first drawback is that all object instances must carry that data even if they do not need it. This requirement could be a deterrent to using the interface for high-volume objects. For example, Doc[1] is a document editor that represents each visible character as a drawable object. These objects are not accessible remotely, as they are part of the document editor's implementation and not available to other applications. We want to define a common interface for drawable objects, but we do not want to burden the character objects with unnecessary overhead.

The second drawback of member data is that it requires virtual base class derivation because in the case of multiple inheritance a class should only contain a single copy of the base class data. As a consequence, if one wishes to allow for the possibility of multiple inheritance then one must use virtual class derivation everywhere.

3.2 Virtual base classes

Martin[4] uses virtual base classes and multiple inheritance in his approach to separating interface and implementation. Our own experience is that virtual base classes are expensive—at least in the C++ implementations that we use. For Fresco, our concern was that requiring virtual base classes would discourage some users.

The potential costs are best illustrated with an example. The code below defines a common base class named Base, two interfaces A and B, where B is derived from A, and implementation classes Aimpl and Bimpl. The A interface defines an operation "f" and the B interface defines an operation "g." Aimpl and Bimpl implement the A and B interfaces, respectively, where Bimpl wants to reuse Aimpl. That is, Bimpl::f() should just be Aimpl::f(), and Bimpl must hold any state needed for Aimpl. In this first version of the code, all derivation is virtual, and Bimpl inherits from both B and Aimpl.

```
class Base {
public:
    Base();
    virtual ~Base();
};

class A : public virtual Base {
public:
    A();
    ~A();
    virtual void f();
};

class B : public virtual A {
public:
    B();
    ~B();
    virtual void g();
};
```

```
class Aimpl : public virtual A {
public:
    Aimpl();
    ~Aimpl();
    void f();
};

class Bimpl : public virtual B, public virtual Aimpl {
public:
    Bimpl();
    ~Bimpl();
    void g();
};
```

An alternative to using multiple inheritance with virtual base classes is delegation, that is, using an object member instead of a parent class and redefining the operations to invoke the corresponding operation on the member. The code below provides the same interface as the previous example, but with the Bimpl class containing an instance of an Aimpl rather than deriving from Aimpl. Bimpl must also redefine A::f to call a_->f().

```
class Base {
public:
    Base();
    virtual ~Base();
};

class A : public Base {
public:
    A();
    ~A();
    virtual void f();
};

class B : public A {
public:
    B();
    ~B();
    virtual void g();
};

class Aimpl : public A {
public:
    Aimpl();
    ~Aimpl();
    void f();
};

class Bimpl : public B {
public:
    Bimpl();
    ~Bimpl();
    void f();
    void g();
private:
    Aimpl a_;
};
```

Note this approach means a Bimpl pointer can no longer be widened to an Aimpl pointer. However, this distinction is not an issue because Aimpl and Bimpl are both implementation classes and therefore hidden from

the application code. Virtual base classes are not necessary even if the interfaces contain multiple inheritance, so long as the generated classes for the interfaces contain no data.

Table 1 shows the cost in memory size of the two approaches compiling with optimization on an Indigo running IRIX 5.1 and a C++ compiler based on Cfront 3.0. The code and data columns are the size in bytes for a file containing the definitions and empty function bodies with the exception of Bimpl::f() in the delegation case, which contains a call to a_->f().

Object	A	B	Aimpl	Bimpl	code	data
Inherit	12	24	24	68	2192	512
Delegate	4	4	4	8	1344	240

Table 1. Code and data sizes in bytes

In this example, delegation is more efficient in terms of memory usage, though less efficient in CPU time because of the extra virtual function call. A sophisticated compiler could, in principle, remove the CPU overhead by noticing that it could fill Bimpl's vtbl entry for f with Aimpl::f and the appropriate offset for the embedded Aimpl object. Regardless, we are willing to trade the CPU time for lower memory usage.

Whether these results are reflective of a particular compiler or applicable to other C++ compilers is not important for our purposes. Our goal is make the use of interfaces convenient and efficient. Since Ix can easily generate either virtual derivation or not, we give the choice of using virtual base classes or delegation to the programmer. Ix also provides a filtering mechanism to make delegation automatic instead of requiring the programmer to code every delegated function. This and other filtering features are described in more detail in Section 4.

3.3 Member functions

Conceptually, the class generated for an interface should have no code as well as no data. However, this approach means that the stubs for remote calls are generated for a subclass, and therefore the stubs for a derived interface must use either virtual base classes or delegation to inherit the stubs for the parent interface. In terms of the previous example, we want the stubs for class B to inherit the stubs for class A.

An alternative approach that we use in Ix is to generate stubs for the functions in the class generated for an interface. For an interface A that defines an operation f, the body of A::f contains the stub to perform a remote call. The stub code performs a virtual function call to access the state necessary to send the object reference to the remote site. The stub object contains a pointer to this state, which must be accessed indirectly anyway to allow different subclasses to use alternate representations of an object reference (it might be desirable to send a copy of the object's state, for example).

4. Filtering

In C++, the signature for every function is written at least twice—once in the class definition and once for the function body. Defining an interface class, whether in IDL or not, adds another definition to this burden. Since we needed an interface translator anyway, we wanted to use the information in the translator to eliminate the burden of repeating function signatures.

We considered several ideas for how to use the translator to generate C++ signatures automatically from IDL. One approach is to generate a file containing empty C++ functions for an interface and let the programmer fill in the bodies. However, this approach does not help if the interface changes after some of the implementation has already been written.

A second approach is to process the implementation files as part of compilation, generating the appropriate function signatures. A separate processing pass is undesirable because it would always slow down compilation, even when the interfaces have not changed. We considered defining C++ preprocessor macros for the signatures, but found that we really wanted to see the expanded signature, not a macro, while we were editing the implementation.

Our approach is to "filter" an implementation header or source file whenever the interface changes. The interface translator first reads the interface definition, then scans the implementation file looking for a line that contains special comments that begin with the characters "//+." Any line not containing these characters is copied as is.

The translator parses the annotations in special comments and generates the appropriate code. For this process to work repeatedly, filtering must eliminate the previously-generated code and copy the annotation comment back out to the new version of the implementation file.

The Ix filtering notation is not intended to be particularly elegant or self-explanatory. Our goals were for filtering to be simple to use, as visibly unobtrusive in the source as possible, and easy to process.

4.1 Class annotations

An implementation class defines some or all of the functions defined on the interface. We use the annotation "interface::op" to indicate that a class definition should include the signature for the given operation. For example, suppose the IDL interface for A is

```
interface A {
    void f();
    long g();
};
```

Before filtering the first time, we could write an implementation class as

```
class Aimpl : public A {
public:
    Aimpl();
    ~Aimpl();

    //+ A::f

    /* other members */
};
```

After the first filtering, the annotation line will appear as

```
    void f(); //+ A::f
```

If the signature for A::f changes, then re-filtering will automatically update the definition in the class. Attributes are similar to functions, except the annotation contains a trailing "=" or "?" to specify the set or get function, respectively.

4.1.1 Generating all operations

A "+" in place of the function name means that the class implements all the functions and attributes defined by the interface. In this case, the translator relies on a line containing only "//+" as an end marker for the lines of generated code. Following the example above, we could have

```
class AnotherAimpl : public A {
public:
    AnotherAimpl();
    ~AnotherAimpl();

    //+ A::+
    //+

    /* other members */
};
```

Filtering the annotation would generate

```
//+ A::+
void f();
long g();
//+
```

As before, re-filtering after changes to the interface will automatically update the class definition. This feature is especially convenient when adding or removing functions to an interface. A "*" can be used instead of "+" to generate inherited functions as well as functions defined by the interface.

4.1.2 Type information for implementation classes

Filtering also provides a mechanism for defining type information for implementation classes. The translator generates type information for interfaces that are used in narrowing and dynamic invocation. Sometimes, one would like to be able to narrow an interface to a specific implementation class. The annotation syntax is "derived : base1[, base2 ...]." For example, if we wanted to narrow to the Aimpl class we could write

```
//+ Aimpl : A
//+
    Aimpl();
    /* other members */
};
```

Filtering this annotation would augment the class specification with type-related member functions

```
//+ Aimpl : A
class Aimpl : public A {
public:
    ~Aimpl();
    TypeObjId _tid();
    static Aimpl* _narrow(BaseObjectRef);
//+
    Aimpl();
    /* other members */
};
```

4.2 Implementation annotations

To define the signature for the body of a function, we use the annotation "C(I::op)" where C is the implementation class name and I is the interface. The translator needs the implementation and interface names because the class definition and implementation may be in separate files and filtering does not perform include file processing.

Continuing the Aimpl example, the initial implementation would be

```
//+ Aimpl(A::f)
{
    /* function body */
}
```

After filtering, the code would be

```
//+ Aimpl(A::f)
void Aimpl::f() {
    /* function body */
}
```

4.2.1 Delegation

The annotation "C(I::+*call*)" generates delegation functions for all the operations and attributes defined on the specified interface. As with class definitions, a "*" instead of a "+" means all operations and all inherited operations.

The "*call*" is the part of the expression for the delegation call excluding the function, but including the member access operation ("." or "->"). For example, if Bimpl is a class that wants to delegate A operations to an Aimpl member "a_" then the filtered code would be

```
//+ Bimpl(A::*a_.)
void Bimpl::f() { a_.f(); }
long Bimpl::g() { return a_.g(); }
//+
```

4.2.2 Controlling support code

The translator also generates support code for the interface classes, including the bodies of the constructor, destructor, and narrow operation. One option is to generate this code in a separate file specified as part of the translation process. We support that option, but we also wanted to give the programmer finer control as to what code is generated and where it is generated.

Filtering makes the custom generation of support code simple. A collection of annotations for an interface specify that certain information should be generated. The annotations are generally of the form "I::%info" where "info" refers to the specific kind of information. The options for info include:

"init" for the constructor and destructor code

"type" for run-time type information

"type+dii" for run-time type information and dynamic invocation support

"stub-externs" for external declarations of stubs for concrete types such as structs

"type-stubs" for stubs for concrete types

"stubs" for remote stubs for operations

"client" for external declarations, type stubs, and operation stubs

"server" for receiving stubs

Several info requests can be separated by commas in a single annotation. An annotation containing only an interface name generates a default set of information, which is currently the initialization and type information.

We use custom code generation in Fresco for two reasons. First, we put the annotations for several interfaces in the same file, reducing the number of implementation files. Second, we avoid generating run-time information for those interface that need not be accessed dynamically, such as the type interface itself.

5. Ix implementation

Ix is about 11,000 lines of C++ code and to date represents roughly half a person-year of effort. Ix parses all IDL constructs, but does not check or generate code for features we have not needed for Fresco, such as contexts.

The Ix implementation is split into the following three main phases:

- Parsing, which is about 3,000 lines of code
- Symbol resolution and semantic checking, which is about 1,500 lines of code
- Code generation and filtering, of which about 3,000 lines are for generation and 1,500 for filtering.

The remaining 2,000 lines implement support data structures and command-line argument processing.

We have been using Ix for the development of Fresco for about six months. Fresco defines about 40 interfaces in IDL and about 100 classes in C++. Currently, about 15,000 of the 35,000 lines in the Fresco library are automatically generated by Ix.

6. Generating documentation

The I2mif program produces FrameMaker Interchange Format (MIF)[2] from comments in IDL source files. The comments are denoted by a line containing either the string "//-" or the string "//.". A line containing the string"//-" followed by additional text denotes the beginning of a definition where the text is the name being defined. A top-level definition ends with a line containing "//-" with no trailing text or a line beginning with a right brace ("}"). A nested definition (such as an operation within an interface) ends at the beginning of another definition or the end of the outer definition.

Lines that begin with "//." indicate text that should be written to the MIF file. Index markers can be specified with the syntax "\marker{text}" where text is the index entry. By default, I2mif creates index entries for names associated with "//-" comments. Formatted text can be specified with the syntax "\emphasis{text}" for italics or "\bold{text}" for a boldface font.

As an example of how this works, here is part of the Transform object interface in IDL:

```
//- TransformObj
interface TransformObj : FrescoObject {
    //. A transform represents a (logically) 4x4 matrix for use in translating coordinates.
    //. A 2-dimensional implementation may store and manipulate a 3x2 matrix rather than
    //. a full 4x4 matrix.

    //- load
    void load(in TransformObj t);
        //. Copy the matrix data from the given transform.

    //- scale, rotate, translate
    void scale(in Vertex v);
    void rotate(in float angle, in Axis a);
    void translate(in Vertex v);
        //. Modify the matrix to perform coordinate scaling, rotation, and translation.
        //. The rotation angle is given in degrees. A 2-dimensional implementation
        //. only implements rotate about the z-axis.
};
```

The formatted output would appear as follows:

TransformObj

interface TransformObj : FrescoObject

A trasnform represents a (logically) 4x4 matrix for use in translating coordinates. A 2-dimensional implementation may store an manipulate a 3x2 matrix rather than a full 4x4 matrix.

load

void load(in TransformObj t);

Copy the matrix data from the given transform.

scale, rotate, translate

void scale(in Vertex v);
void rotate(in float angle, in Axis a);
void translate(in Vertex v);

Modify the matrix to perform coordinate scaling, rotation, and translation. The rotation angle is given in degrees. A 2-dimensional implementation only implements rotate about the z-axis.

Before producing the output interchange file, I2mif sorts all the top-level definitions (interfaces) by name and within each top-level definition also sorts the nested definitions by name. The effect is a "dictionary" of in-

terfaces and operations that is generated automatically from the IDL source. Using I2mif has help tremendously in keeping the interface documentation and source in sync with each other.

7. Conclusions

Separating interface from implementation is an important part of building a software system. Acceptance of using interfaces will depend to a large extent on the ease with which interfaces can be defined and modified.

Ix is a flexible tool for translating CORBA IDL to C++ that we have developed as part of the Fresco project. Our experience with Ix has been that implementation filtering makes programming with interfaces as easy or easier than straight C++. The ability to implement delegation easily has also made it simple for us to avoid the use of virtual base classes.

Overall, our software is more abstractly specified, more powerful, easier to edit, and as efficient as if we had not separated interface and implementation. Rather than spending time on mechanics, using Ix allows one to concentrate on the semantics of interfaces and their possible implementations.

8. Acknowledgment

Steve Churchill implemented I2mif.

9. Availability

Fresco will be available as part of the X11R6 distribution in the spring of 1994, including the source for Ix, run-time library, and the Dish application. This software will be available without restrictions on use. For further information about obtaining Fresco, send electronic mail to linton@sgi.com.

10. References

[1] P. Calder and M. Linton. The Object-Oriented Implementation of a Document Editor. *OOPSLA '92*, Vancouver, British Columbia, Canada, pp. 154-165.

[2] Frame Technology Corporation. *Maker Interchange Format (MIF) Reference Manual.*

[3] M. Linton and C. Price. Building Distributed User Interfaces with Fresco. *Proceedings of the Seventh X Technical Conference*, Boston, Massachusetts, January 1993, pp. 77-87.

[4] B. Martin. The Separation of Interface and Implementation in C++. *Proceedings of the USENIX C++ Conference*, Washington, D.C., April 1991, pp. 51-63.

[5] Object Management Group. Common Object Request Broker Architecture and Specification. OMG Document Number 91.12.1, Revision 1.1.

[6] J. Ousterhout. Tcl: an Embeddable Command Language. *Proceedings of the 1990 Winter USENIX Technical Conference.*

[7] B. Stroustrup and D. Lenkov. Run-Time Type Identification for C++ (Revised). *Proceedings of the USENIX C++ Conference*, Portland, Oregon, 1992, pp. 313-339.

A Poor Man's Approach to Dynamic Invocation of C++ Member Functions

Thomas Kofler, Walter Bischofberger, Bruno Schäffer, André Weinand
Union Bank of Switzerland
UBILAB (Information Technology Laboratory)
Bahnhofstr. 45, CH-8021 Zürich, Switzerland
{kofler,bischofberger,schaeffer,weinand}@ubilab.ubs.ch

Abstract

During the last year we built several solutions for opening our ET++ applications for internal and external scripting. The most annoying part to be coded manually was the code stubs that translate a string based request into the invocation of a member function.

For this reason we built an ET++ specific solution that provides dispatchable member functions in an inexpensive, non-intrusive way. Our solution consists of an extension of the macro generated ET++ run time meta information. To make a member function dispatchable, a developer has to write one macro call. This generates a member function meta object providing information about arguments and a function that serves to invoke the respective member function. These two generated parts work in the context of the dynamic invocation framework, which embodies an architecture that can be customized for varying interfacing needs.

1. Introduction

Our team has been developing interactive standalone applications based on the ET++ application framework for years. During this time the standard application model embodied in ET++ [Wei88, Wei89] fulfilled our requirements well. During 1993 we started to develop distributed applications where several services and tools cooperate to provide a group of users with a certain functionality. A typical example is Beyond-Sniff, a platform for cooperative multi-user development environments. It provides a set of cooperating services and tools. In developing Beyond-Sniff we found that we needed a conceptual framework for interpretively driving an application from within, as well as from other applications. The latter could be patterned, for instance, after AppleScript [App93].

Once we had identified this requirement it was an obvious approach trying to enhance our application framework in order to provide this functionality in a generic way. Our first approach is described in [Kof93]. In this approach we manually wrote stubs to dispatch internal and external dynamic member function invocations. While this proved handy for the first prototypes, it became more and more annoying in large scale applications. For this reason we decided to implement a generic solution that works within our application framework.

A generic solution has to provide a way to invoke an operation whose name is determined at run-time and to provide it with arguments (something straightforward in dynamic languages as Smalltalk or Lisp).

We intended to find a solution that:

- is open and can be used by different kinds of clients, such as embedded interpreters, or by an interapplication scripting mechanism,
- is simple, declarative, and requires little programming from the programmer,
- is type-safe
- is non-intrusive, i.e., the existing code of the application framework must not be changed,
- does not affect portability of the ET++ application framework,
- can be easily implemented and maintained,

- and does not require any additional tools, such as preprocessors or parsers.

The resulting dynamic invocation framework, which we present in this paper, is a typical case of a poor man's approach that works well for its intended application area but also has its limits.

The goal of this paper is to explain the dynamic invocation framework in detail. Section 2 shows how the dynamic invocation framework is used. Section 3 presents the architecture of the framework. Sections 4 and 5 discuss what kind of member functions are supported. Section 6 explains the conversion process between C++ and external data representations. Section 7 compares the dynamic invocation framework with similar approaches, and in Section 8 we draw our conclusions.

2. Using Dynamic Invocation

Dynamic invocation of member functions has many useful applications. In this section, we present concrete examples. We also sketch how the programmer uses our dynamic invocation library without going into details.

Suppose a programmer has written a graphical editor in C++. There is a class, say 'Editor', that has a set of member functions, each of them representing a command available to the user. The user invokes command by selecting an item from menus. There are several ways to connect menu items with their respective member functions acting as entry points. Attaching a string containing the name of the respective member function of the class Editor is simple, flexible, and straightforward if member functions can be dynamically invoked. The code in example 2.1 demonstrates the idea. Note that the function 'CallMemberFunction()' is a simple interface to the dynamic invocation dispatcher.

```
class Editor : public Tool {
public:
    int CallMemberFunction(const char *cmd);

    virtual int Close(bool askUserIfChanged);          // (1a)
    virtual int Save();                                // (1b)
};
Msg1(AT(int, Out, --), Editor, Close, AT(bool, In, --));   // (2a)
Msg0(AT(int, Out, --), Editor, Save);                      // (2b)
....
```

```
menu->Add(new MenuItem(cLabelClose, "Close true"));    // (3a)
menu->Add(new MenuItem(cLabelSave, "Save"));           // (3b)
...
editor->CallMemberFunction(menuItem->GetCommand());    // (4)
```

Ex. 2.1 Connecting Menu Items to Member Function by Dynamic Invocation

If the programmer uses our dynamic invocation library, he writes the macro calls as shown in example 2.1 to make 'Editor::Close(bool)' and 'Editor::Save()' available to dynamic invocation. These declarations turn member functions into *dispatchable member functions*.

Embedding an interpreter for user-level scripting is another interesting area where dynamic invocation is helpful. Manually implementing the code that provides application-specific functions in the interpretive language is tedious and error-prone. Dynamic invocation of member functions may save a lot of time and assures that argument conversion is done automatically and consistently.

If applications can send commands to each other, they can be integrated to any desired degree. Again, an application that can process commands sent via interprocess communication requires code that transforms a command in an external form into an invocation of a function written in the native programming language.

3. Architecture of the Dynamic Invocation Framework

The macro calls as shown in example 2.2 expand to an ordinary static function, called the *callee stub*, and to a statically allocated object, called the *member function meta object*. The primary purpose of the function is to wrap the invocation of the member function, and triggering argument conversion as explained below. The member function meta object, or function meta object for short, describes the argument list of the member function, and also contains a pointer to the callee stub wrapping the member function. The pointer to the function meta object, along with its name, is stored in the class descriptor object. This is shown in figure 3.1.

A *class descriptor* is an object that describes a C++ class. An ET++ object can be asked for its class descriptor by invoking the member function IsA(). The most important use of class descriptors is dynamic type checking. ET++ class descriptors have been used for some time. They are defined by means of a macro much like function meta objects. See [Gam89] for details.

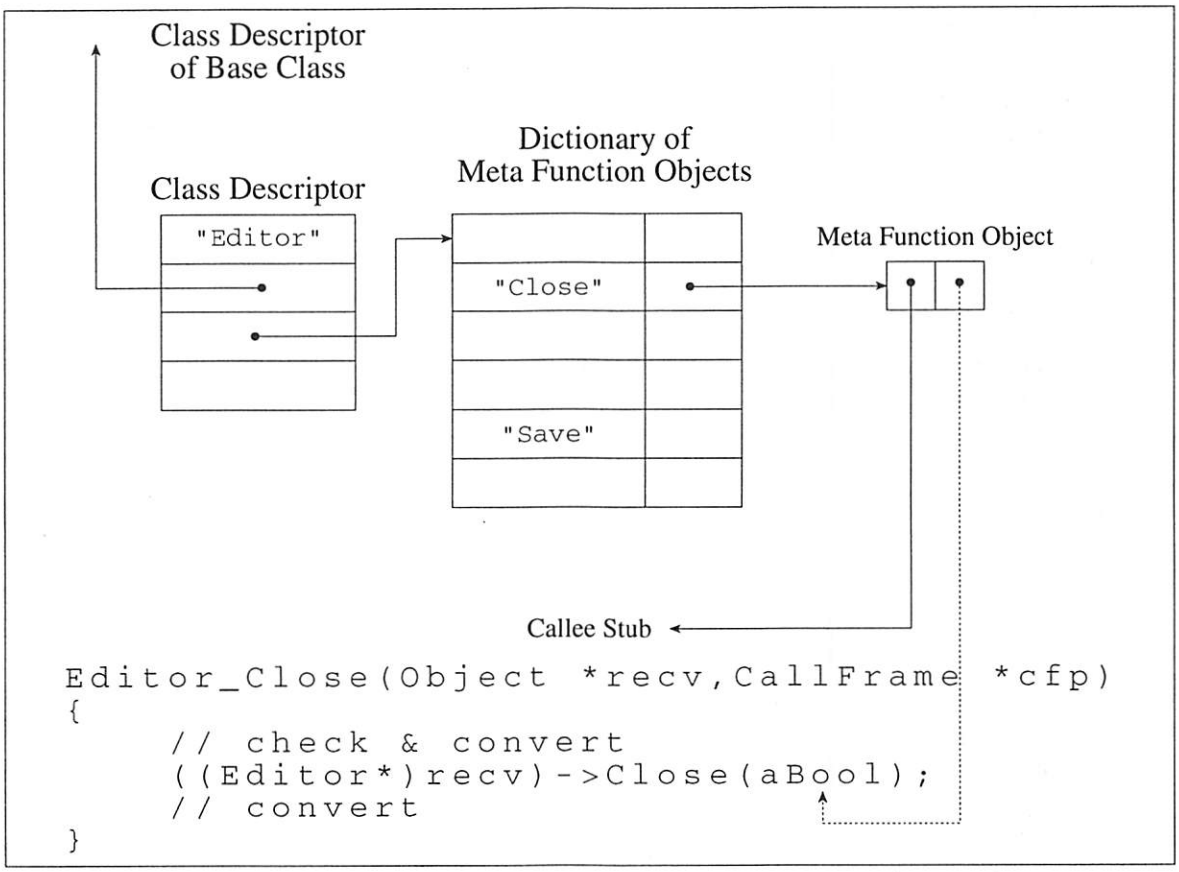


Fig. 3.1 Structure of the Meta Information for Dispatchable Member Functions

The callee stub also passes the member functions its arguments. The conversion is done by an object known as *call frame*. A call frame object knows the meta function object and the external representation of the invocation, e.g. a string. The call frame interprets the description of the argument list as provided by the function meta object and converts, driven by the argument list description, the arguments from the external representation. After the member function has returned, the return value is converted into a form as

required by the particular external representation, and memory allocated by the call frame before the invocation is deallocated.

Two components remain to be introduced. The *dispatcher* is responsible for looking up the meta function object and for creating a call frame that can deal with the particular external representation used. The dispatcher then invokes the callee stub passing it a pointer to the object the member function is invoked for, and an appropriate call frame. If a member function involves pointers to ET++ objects as an argument or as return value, then the call frame contacts the *object name mapper*. The object name mapper maintains a map that associates names with object pointers.

Since we build on the existing meta information provided by ET++ our approach works only for instances of classes derived from the ET++ class Object. By porting or reimplementing the meta information infrastructure this restriction could be eliminated. Second, the kind of arguments that can be passed is restricted. This topic is discussed in depth in Section 4 and 5.

4. Supported Signatures

In this section, we give an overview of what our dynamic invocation framework can cope with. We cover the fundamental and actual limitations, and we discuss why these limitations do not pose problems as long we do not leave the realm of the ET++ class library.

We have two dimensions regarding the range of member functions that can be made dispatchable. One of them is the range of applicable types. The other is the range of supported calling semantics. In this section, we concentrate ourselves on applicable types. For a complete understanding, section 4 and 5 must be viewed as a unit.

Overview

We start by introducing some technical terms. In C or C++, the type of an argument or a return value consists of a *base type*, and optionally, of a *type constructor*. A type constructor is a *type expression* using *type operators*. Type operators are the dereferencing operator ('*'), the address operator ("&"), and the subscript operator that is denoted as pairs of brackets with or without an integral constant enclosed[1]. Defining a new C++ class means to introduce a new base type whereas expressing a type by means of an existing base type and a type expression does not introduce a fundamentally new type. Notice that a typedef name is not a base type in our sense, but an alias for a type expression.

The applicable set of base types and type constructors is constrained for our dynamic invocation framework. Base types have to satisfy some conditions discussed below whereas the set of applicable type constructors is small and fixed. The limitations regarding base types and type constructors are of different nature. The set of applicable type constructors is fixed insofar that we have to adapt the dynamic invocation framework if a new type constructor is to be supported.

The set of supported C++ type constructors contains pointers or one-dimensional vectors, both involving up to three levels of indirection. There is a well-known twist with pointers and vectors in C++ since there is no syntactical difference between them. In section 4, we will explain how we handle type constructors in detail.

We support the following base types: all primitive C++ types (char, int, double, etc.), classes like Point and Rectangle viewed as though they were *built-in* into C++, and the universal class 'Object'. The class Object is the root of the ET++ class library. Supporting the class Object automatically supports all its derived classes. A built-in class is a class whose objects are normally passed by value. Objects of built-in classes do

---

[1] The function call operator '()' is also a type operator in our sense, but pointer to functions are not supported anyway.

not have a real identity, and they are normally defined as automatic object variables. In contrast, ET++ objects are never passed by value, and they are always allocated from the free store.

Introducing a new base type playing the role of a new built-in does not pose difficulties. The same is true if we introduce a new ET++ class. In both cases, a programmer defines a simple meta object, or a class meta object. The definition of a meta object describing a built-in or an ET++ class is a simple and inexpensive task. The presence of the meta object suffices to use the base type in the signature of a dispatchable function.

Typedef names are not base types in our understanding. A *trivial typedef* name is an alias for another base type. A *non-trivial typedef* name is an alias for a general type expression. We cannot support any type expression that takes a non-trivial typedef name for a base type because we cannot resolve, unlike the C++ compiler, the typedef name.

Enumeration types currently are not supported. This is a limitation that could be alleviated. What we have to do is to extend the infrastructure for defining enumeration meta objects.

ET++ Signatures Use Only Few Types

We will now discuss why our dynamic invocation framework, despite its limitations, proves to be useful. An examination of the signatures of the member functions found in ET++ classes shows that we have only a small number of base types, and a small number of type constructors. Since almost all ET++ classes are derived from the root class Object, supporting the class Object automatically includes the support of all its derived classes.

All member functions in ET++ have comparably simple signatures. The fundamental reason behind this fact is that most concepts are encapsulated in ET++ classes. For example, complex data structures are usually constructed with container objects. Thus, a simple pointer to the anchor of a complex data structure suffices to refer to the data structure. As a consequence, member functions never take arguments with a complicated structure. Argument types are rather simple, like pointers to objects of any complexity, integers, or pointers to integers, etc.[2]

The range of supported base types and type constructors was derived from inspecting the signatures of the public ET++ member functions. Even if the protocols of ET++ classes would radically change in the future, we do not expect any changes regarding the set of supported base types and type constructors.

5. Member Function Meta Objects

A member function meta object describes its member function. Function meta objects are stored on per class basis in a dictionary that enables for lookup by the name of the respective member function. A function meta object contains:

- a reference to its corresponding class meta object
- the external name of the member function
- a description of the member function's signature
- a function pointer to the callee stub.

Defining Function Meta Objects

Function meta objects are denoted by means of macros resolved by the C++ preprocessor. Similar macros are used for denoting class meta objects. There is conceptually one macro named 'Msg' to define a function meta object. Since the preprocessor does not support macros with variable argument lists, we have a family of macros with the same name, but with different suffixes designating the number of arguments.

---

2 Interestingly, this is also true for the types of data members. Otherwise, the meta object approach as described in [Gam89] would fail for the same reason.

The macro call denoting the function meta object supplies all needed information: the class the function is member of, the name of the member function, and a description for each argument including the return value. The template in example 5.1 shows the syntax.

```
Msg N(ReturnValueDescription, Class, MemberFunction, ArgumentDescription1, ...);
```

Ex. 5.1 Syntax Template of the Msg *N* Macro

An argument or the return value is described using a macro again. The name of this macro encodes the type constructor. The first argument is the base type to which the type constructor is applied. Its second argument specifies the direction of the information flow (also known as mode), and the last argument describes who owns the memory involved with an argument. Some of the argument description macros take four arguments. The additional argument to the macro describes what arguments of the member function contains size information about vectors of variable size. We will show concrete examples covering this case later in this section. The template in example 5.2 shows the syntax.

```
TypeConstructor(BaseType, Mode, OwnershipDescription, SizeInformation);
```

Ex. 5.2 Syntax Template of an Argument or Return Value Description

Conveying Structural Information about Pointers and Vectors

The type constructor encoded in the macro's name gives more information than its corresponding C++ notation. Using different type constructors for syntactically equivalent notations conveys structural information. This information answers the question what 'Object **' means: Is it a pointer to a pointer, or is it a vector of pointers, or is it a vector of vectors of Objects? Except for the first case, we also need to know how many slots the vectors on the second and third levels have.

Consider the code in example 5.3. The different type constructors used in the example tell us how a function interprets arguments involving pointers or vectors. For instance, the type constructor of U's second argument describes it as a pointer to a pointer (=PP), i.e., that the argument conceptually is a pointer passed by reference. For the second example V, we have a vector of pointers (=ZP). The vector is null-terminated. Finally, W also takes a vector of pointers, but this time its size is passed in the argument named size. The reader might have noticed that the macro AVP takes four arguments. The fourth argument specifies what arguments of the member functions carry size information for the argument being described. The actual argument for Size(), here the number one, specifies the relative position of the argument containing size information. Thus, the third argument of W contains the actual size of the vector.

```
void Foo::U(int i, Object **);                    // (1)
void Foo::V(int i, Object **);                    // (2)
void Foo::W(int i, Object **, int size);          // (3)

Msg1(Void, Foo, U, AT(int, In, --), APP(Object, InOut, --));                    // (1)
Msg1(Void, Foo, V, AT(int, In, --), AZP(Object, In), ()));                      // (2)
Msg2(Void, Foo, W, AT(int, In, --), AVP(Object, In, --, Size(1)), AT(int, In, --));  // (3)
```

Ex. 5.3 Same C++ Type with Different Meanings

Argument Modes

The second argument to a type constructor macro describes the *mode* of a member function's argument. The mode argument can take the following values: In, Out, or InOut. These values indicate the direction of the information flow.

In the case of an argument with In mode, information flows into the function, but not back. In the case of an Out argument, information flows out of the function. We assume that the data structure referenced by an In argument is fully initialized before calling the member function, whereas the data structure referenced by an Out argument is not initialized at all. The data structure of an InOut argument also is fully initialized, but we are interested in the values stored in the structure after the call, too.

Ownership Description

The mode of an argument does not convey enough information to determine what memory has to be allocated before the invocation, and what memory has to be deallocated after invocation. The third argument to a type constructor macro call carries information such that we can deduce, within the bounds of our approach, the ownership of memory. The argument is called the ownership description.

Consider the signature 'char *Foo::Name()'. Obviously, the return value of Foo::Name is an Out argument. The question is whether the string has been allocated on behalf of the caller, or whether the string is still used by the callee after the invocation. Since the latter is a common case, additional information about ownership allows to cover a larger range of calling semantics.

The ownership description is based on the levels of indirection involved, and on the silent assumption that memory for a level belongs to the caller if the opposite is not stated. For instance, the string referenced by an argument of type 'char *' is on the first level of indirection, the strings referenced by an argument of type 'char **' are on the second level, and so forth. If memory referenced, say, on level 1 belongs to the callee, we describe this fact by writing 'Mine(1)'. Ownership descriptions can be combined by concatenating them by means of the dot. Example 5.4 shows two typical examples where memory referenced by a return value is not allocated on behalf of the caller.

```
char *Foo::Name();
char **Foo::GetArgv();

Msg0(AZ(char, Out, Mine(1)), Foo, Name);
Msg0(AZZ(char, Out, Mine(1).Mine(2)), Foo, GetArgv);
```

Ex. 5.4 Describing Ownership of Memory

For ET++ objects, ownership descriptions are meaningless. If a pointer to an ET++ object is expected as part of an In argument, we map the object name to the object pointer. If a pointer to an ET++ object is referenced by an Out argument, then the call frame tells the object name mapper to generate a new name if necessary.

6. Call Frames and Callee Stubs

This section discusses call frames and how they relate to callee stubs. The Msg*N* macros discussed in section 5 not only expand to code that defines the statically allocated function meta object, but also code for the callee stub. A callee stub always has the signature as shown in example 6.1.

```
int SomeClass_Fct(SomeClass *receiver, CallFrame *ffp);
```

Ex. 6.1 Signature of Callee Stub

The callee stub is invoked by a dispatcher which is also responsible for providing the pointer to the receiver object[3], and an appropriate call frame object. The call frame object is used in the callee stub to provide the

3 the 'this' pointer

member function call with arguments. The call frame has the actual arguments at its disposal in form of the external representation, e.g., a string with a particular syntax.

To invoke the member function, conversions are needed to transform values from the external representation to the C++ representation, and from the C++ representation to the external representation. A call frame object is responsible to do all these conversions. For accomplishing the conversions, the callee stub calls some member functions of the call frame to obtain the actual value of an argument. Of course, the macros presented in section 5 and the call frame's class interface are designed to smoothly work together.

A call frame needs the information contained in the member function meta object. The reason is to minimise the number of access functions we need to provide in the call frame's public interface. Selection of the appropriate conversion function is not done by name, but by exploiting the C++ matching mechanism for overloaded functions. This means, as the most important implication, that we need exactly one conversion function for all pointer to ET++ objects. There is no need to change the set of conversion functions if a new ET++ class is introduced. If we had to do so, the whole approach would be of little use.

The callee stub is coded such that its wrapped member function is not invoked if one of the argument conversions fails. Conditions of this kind are signalled to the dispatcher by the return value of the callee stub. In any case, the call frame is given the chance to do the necessary cleanup, i.e., to possibly deallocate memory belonging to the caller.

It is worthwhile to notice that type errors in the code generated by the function meta object definition macro are detected at compile time. Wrong specifications of mode, ownership, or size, cannot not be detected, of course.

7. Comparison with Similar Systems

In this section we compare our approach with CORBA [OMG91, OMG92] and MetaFlex [Joh93]. We chose CORBA because it is an emerging standard for distributed objects. This comparison is similar to comparing an aircraft carrier with a little boat, but it reveals interesting insights. MetaFlex, on the other hand, was developed with goals similar to ours but its developers ended up with a completely different approach.

CORBA

CORBA was designed with the idea of creating a standard for interacting with distributed objects. One of our goals was to find an inexpensive non-intrusive solution for (remotely) interacting with ET++ objects.

CORBA provides a traditional object model that can be mapped to a wide number of programming languages which do not even have to be object-oriented. Our object model is defined in C++, and it is restricted by the way C++ is used in ET++.

CORBA defines two ways how clients can interface with the Object Request Broker. Compiled clients use generated code stubs for invoking member functions of distributed objects. Interpreted clients use the dynamic invocation interface to pass an invocation request to the Object Request Broker. In our solution we cannot provide for interfacing via client stubs. Our intended clients are interpreters that use a dynamic invocation interface.

Every CORBA object request broker provides one single space for all its clients and makes it transparent in which server a certain object resides. In our solution every server has its own object name mapper. Object names are therefore only valid in the context of a server.

CORBA was designed is intended to provide for interoperability over different platforms of different vendors. Our solution is intended for scripting ET++ applications running on different platforms.

MetaFlex

MetaFlex was developed to accelerate the implementation of AppleScript support for applications implemented with Aldus' application framework. MetaFlex has therefore almost the same purpose as our dynamic invocation framework.

MetaFlex parses C++ class definitions to generate the meta information available at run time. The output of MetaFlex is C++ code which is compiled and linked with the code of the classes it describes. In order to reduce the amount of code that is generated MetaFlex provides a language that serves for specifying the kind and amount of meta information to be provided.

Compared to MetaFlex our approach is lightweight and inexpensive. MetaFlex is, however, a much more powerful system because it does not need to restrict the kind of C++ code that can be processed. MetaFlex allows a developer to control what kind of meta information is needed. Our approach merely permits a developer to specify which member functions are dispatchable.

The generality of MetaFlex has its price. Its implementation was no trivial endeavour whereas implementing our dynamic invocation framework took a relatively small effort. We could also profit from previous work such as the infrastructure for ET++ class meta objects [Gam89]. Johnson et alii also reported that they had to adapt the parser because it could not deal with code not compliant with the standard. We avoided such problems by letting the C++ compiler do the work.

MetaFlex as described in [Joh93] is neither portable, nor is the mechanism for dynamic invocation type-safe. Our dynamic invocation framework is type-safe and portable. The mentioned deficiencies are not inherent to the MetaFlex approach, however.

8. Conclusions

In this paper, we presented our dynamic invocation framework for string based dispatching of C++ member functions, and how we applied the framework when we embedded the scripting language Tcl into ET++. Our approach has proved powerful enough to endow the application framework ET++ with an infrastructure that supports scripting in general. It makes the tedious and error-prone work of writing callee stubs superfluous. The solution we described virtually eliminates all manual work.

There are different ways how the functionality for dynamically invoking C++ member functions can be achieved. We decided to sacrifice generality in favour of an inexpensive, but limited solution. Our approach works well only for a single-rooted class library like ET++. A solution like MetaFlex, which is based on a C++ parser, is general, but expensive in terms of development time.

A developer using an application framework with scripting support profits in two respects. First, he or she does not need to start from scratch to make an application scriptable. The basic functionality is rather inherited. As a welcome side effect, the scripting interface to the generic components of the application framework automatically remains the same. Second, the application framework offers a set of useful components that reduce the amount of work to make the specific functionality available for scripting.

Scripting on the interapplication level becomes increasingly important. If we use C++ to implement many applications, we need to overcome the static nature of that language. Supporting dynamic invocation of member functions is the key to solve the problem. As long as we use the same application framework for developing a number of applications, a specific approach like ours is feasible, sufficient and economically interesting.

A. References

[Appl93] Apple Computer, Inc.: AppleScript Developer's Toolkit Version 1.0 (Part Number 030-3994-A). Cupertino (CA), 1993.

[Bec88] Kent Beck, William Cunningham: A Laboratory For Teaching Object-Oriented Thinking. In: OOPSLA'89 Proceedings, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 372-377.

[Gam89] Erich Gamma, André Weinand, Rudolf Marty: Integration of a Programming Environment into ET++ - a Case Study. In: Proceedings ECOOP'89, Cambridge University Press, Cambridge (Nottingham, UK), 1989, pp. 283-297.

[Kof93] Thomas Kofler: Integrating Interpreters in the Application Framework ET++. Student Term Project in Computer Science, Institut für Informatik, Universität Zürich, 1993.

[Joh93] Richard Johnson, Murugappan Palaniappan: MetaFlex: A Flexible Metaclass Generator. In: Proceedings ECOOP'93, Springer-Verlag, July 1993, pp. 502-527.

[OMG91] Object Management Group: The Common Object Request Broker: Architecture and Specification. Revision 1.1 (OMG Document 91.12.1), 1991.

[OMG92] Object Management Group: Object Management Architecture Guide. 2nd Edition (OMG Document 92.11.1), Framingham (MA), 1992.

[Ost90] John K. Osterhout: Tcl: An Embeddable Command Language. In: Proceedings of the USENIX Winter Conference, Jan 1990, pp. 133-146.

[Wei88] André Weinand, Erich Gamma, Rudolf Marty: ET++ - an Object-Oriented Application Framework in C++. In: OOPSLA'88 Conference Proceedings, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, November 1988, pp. 168-182.

[Wei89] André Weinand, Erich Gamma, Rudolf Marty: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. In: Structured Programming, Vol. 10, No. 2, June 1989, pp. 63-87.

Sharing Between Translation Units in C++ Program Databases

Samuel C. Kendall
Sun Microsystems Laboratories, Inc.
2 Elizabeth Drive
Chelmsford, MA 01824
sam.kendall@east.sun.com

Glenn Allin
CenterLine Software, Inc.
10 Fawcett Street
Cambridge, MA 02138
glenn@centerline.com

ABSTRACT

A C++ program database represents information about C++ code, typically to enable program browsing or debugging. Such databases can grow very large. The growth is fundamentally due to the *translation unit* (*TU*) program structure C++ inherited from C: a naively designed database will consist largely of representations of redundant or unused code from header files.

This paper measures the effect of some techniques for shrinking this naively designed database: the *elision* of unused entities from a TU, and the *sharing* or *linking* (generically, the *combination*) of redundant entities across TUs. We also measure the overhead imposed by the *segregation* of class types with external linkage from those with internal linkage.

We define and measure these techniques for our own database, which was designed with very specific requirements. We also discuss techniques and organizations used in other program databases to save space: sharing at header file granularity; ruthless simplification of the database; and lazy loading of data into the database. Finally, we note the potential problems associated with independently implemented translators feeding into the same database.

1. Introduction

A C++ program database represents information about C++ source or object code, typically to enable program browsing or debugging. Such databases, if they do not take steps to avoid it, can grow very large. This growth is fundamentally due to the *translation unit* (*TU*) program structure C++ inherited from C: a program is a set of TUs, and declarations are shared between several TUs by placing them in header files and textually including the header files (by means of `#include` directives) in each TU. In C++ programs, broadly speaking, most of the source code a translator sees comes from header files. Much of this code is redundant, because a given header file is included in many TUs, and often much of the included code goes unused in any given TU that includes it. A naively designed database will consist largely of representations of this redundant and unused code, wasting time (the time needed to store the extra code) as well as space.

This paper measures the effect of two techniques for shrinking this naively designed database: the *elision* of unused entities from a TU, and the *sharing* or *linking* (generically, the *combination)* of redundant entities across TUs. We also measure the overhead imposed by the *segregation* of class types with external linkage from those with internal linkage; segregation turns out to be necessary to maintain semantic consistency.

To measure these techniques we employed a modified version of the ObjectCenter component debugging engine, which we call "our database". The component debugging engine is part of CenterLine Software's ObjectCenter C++ programming environment [Wyant]. Although the techniques we discuss are general, the results of our measurements are of course specific to our database. To place our measurements in context and to allow the reader to judge if they might apply to some other database, we explain the organization of our database in section 2.

Sections 3, 4, and 5 define elision, combination, and segregation respectively, and discuss these techniques. Section 6 measures the effect of these techniques on three medium-sized C++ program fragments and draws conclusions from the measurements. Section 7 discusses related work, including alternate techniques and database organizations that lead to smaller program databases. Section 8 concludes the paper.

Much of this paper is about class types. Many of the points about class types, we should note, also apply to enumeration types; for brevity we rarely mention enumerations explicitly.

2. Our Database

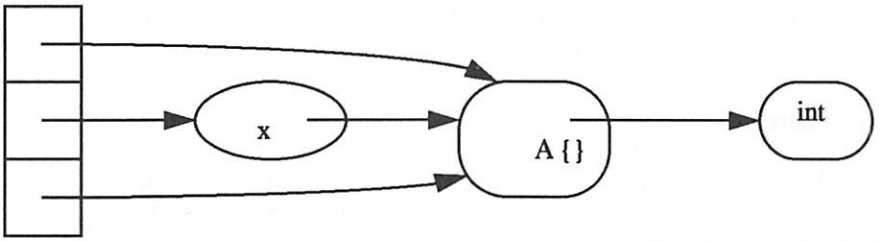
The ObjectCenter component development engine database supports browsing, debugging, interactive translation (to interpreter bytecodes) and execution of source code fragments, mixtures of object code and interpreted code, incremental TU linking, and mixtures of C and C++. The database is kept entirely in virtual memory. Tools access the database through a message-passing interface.

For simplicity of explanation, we do not discuss the entire database in detail. The portion of the database we discuss is sufficient to support browsing, interactive translation of source code fragments, incremental TU linking, and C++ only. Most of our code examples are confined to a subset of C++ containing only top-level variables, functions, and classes. We do measure (in section 6) a larger portion of the database than we discuss in detail.

Our C++ program database represents a program as a set of translation units (TUs); each TU is a sequence of declarations, and each declaration refers to an entity such as a class type, a variable, or a function. For example:

```
struct A;
extern A x;
struct A { int i; };
```

We represent the database that results from this TU as a diagram:



Boxes represent declarations. Ovals represent variables. Rounded rectangles represent types. Arrows represent relationships between entities. The "{}" after "A" indicates that A is defined.

To keep the diagram simple, we have omitted much: the name of `A::i`; that the first declaration of A is a forward declaration, while the second is a definition; that `x` is only extern-declared as opposed to defined. Also missing are tables that allow fast lookup of entities by name or by other attribute. These things, and others, are in our database. But the diagram captures the essentials we need for our discussion.

Note that if there is more than one declaration or definition for an entity in a TU, there is only one representation for that entity in the representation of the TU. For example, there is only one representation of `A` above, even though there are two declarations of that class. Representing declarations rather than declared entities can make sharing between TUs easier, as we discuss in section 7.2. Our database has the "one representation per TU" organization because that is what our C++ front end produces and consumes. CenterLine's design philosophy has been to alter the C++ front end (which is supplied by a third party) as little as possible.

In the following sections we view our database as a directed graph; when we refer to "a graph" in this paper, we mean a single TU, or a database, considered as a directed graph. Nodes have attributes, and edges connect nodes. We do not give a complete list of the different kind of nodes, attributes, and edges; but the following abridged list should give an idea of what we mean:

A TU is a sequence of declarations. Each declaration has a Boolean attribute "is-a-definition" and an edge to the entity it declares. There are many different kinds of declared entities, among them top-level variables and class types.

Each top-level variable has a name attribute (a character string), a Boolean attribute "is-a-definition", a linkage attribute ("external" or "internal"), and an edge to a type.

Each class type has a name attribute, a Boolean attribute "is-a-definition", a Boolean attribute "is-a-structure" (if false, the class is a union), a linkage attribute ("external", "internal", or "none"), a sequence of edges to base nodes, a sequence of edges to friend classes and functions, and a sequence of edges to members. A base node has an access control attribute ("private", "protected" or "public"), a Boolean attribute "is-virtual", and an edge to a class (the base class).

There are many kinds of members. Data members, for example, have a name attribute, an access control attribute, a Boolean attribute "is-static", and an edge to a type.

Our diagrams of databases are simplified versions of these directed graphs, with only the most nodes, attributes, and edges pictured.

A TU is added to a database in the following manner. First, the source code for the TU is translated into a stand-alone graph. Second, this graph is added to the graph which is the database. We call these phases *TU translation* and *TU combination*, respectively; they correspond to compilation and linking in a traditional system.

The next three sections describe elision, combination, and segregation. The scenarios we measure are determined by whether each of these techniques is invoked during TU combination (actually, elision is invoked just before TU combination). Algorithms to implement these techniques are beyond the scope of this paper, but we have taken pains to accurately define the terms (elision, sharing, linking, and segregation) that capture the effect of our algorithms.

3. Elision

Informally, elision means "omit everything from a TU that is not used". To define elision more precisely, we define first a source-to-source transformation on TUs:

> *Elision transformation*: the omission of all non-defining declarations, and the omission or demotion to forward declarations of all class definitions and inline function definitions, unless the omission or demotion would render the code ill-formed or would remove a member of a class.

For example:

TU before elision trans.	TU after elision trans.
`struct A { int i; };`	`struct A;`
`struct B { A* ap; };`	`struct B { A* ap; };`
`struct D : B {};`	`struct D : B {};`
`extern A a;`	
`void f();`	
`D d;`	`D d;`

From bottom to top: `d` is kept because it is a definition. `f` and `a` are discarded because they are unused. `D`'s definition is kept because `d` depends on it, and `B`'s definition is kept because `D` depends on it. `A`'s definition is demoted to a forward declaration. `A` is not eliminated entirely because `B`'s definition depends on its forward declaration.

Elision, then, is the discarding of part of a graph as though the elision transformation had been applied to the source code. Elision is a filter on graphs, applied just before TU combination.

We define elision in terms of a source-to-source transformation as a way of preserving invariants the violation of which would discomfit a database client. For example, a more aggressive elision algorithm might elide the body of `B` above, even though `B` is a base class of `D`; and we could enable our database to store the odd result, a class (`D`) with a "base class" edge to an undefined class (`B`). But class browsing tools, many written before elision was implemented, will be surprised to find a base class with no definition; some of these tools may crash when confronted with the undefined base class.

A happy property of elision is that an entity is completely missing from a database only if it is used in none of the TUs. For example, suppose a particular type is declared in ten TUs, but only used in two of them. Even though that type may be elided from eight TUs, it will still be present in two of the TUs, and thus it will be in the database and available to the user.

In our experience elision is acceptable for a debugging database. Occasionally the declaration of an entity the debugger user wants to access has been elided; but usually this is not the case, due to that happy property.

For a browsing database, however, elision may not be acceptable [Grass]. Cia++ [Grass & Chen], for example, can be used to discover unused code or needless header file inclusions. Elision would destroy the data needed to make these discoveries.

There are no C++ language issues with elision—elision was carefully defined to avoid them—except that one must be careful to compute the linkage of a type before performing elision. In this example, the elision transformation changes class `A`'s linkage from external to internal:

```
// TU before elision trans.    // TU after elision trans.

struct A { int i; };           struct A { int i; };
void f(A);
static A a;                    static A a;
```

Our implementation incorrectly computes the linkage of a class after elision. We expected this bug to generate many classes with falsely internal linkage, but in practice this did not occur very often. In most cases where all externally linked uses of a class were elided, the class itself was elided or at least demoted to a forward declaration. Linkage is discussed in section 5.

4. Combination

Combination is the merging of nodes in the graph in order to reduce the number of nodes. Note that "combination" is different than "TU combination"; TU combination (the inserting of the graph for a TU into a database) may be performed with combination (the merging of individual entities) or without it.

The word "sharing" is used loosely in the title of this paper. More precisely, this paper is about *combining* representations of C++ language entities between TUs. Sharing is one mode of combination.

We define two modes of combination:

> *Sharing*: two entities are sharable if their attributes (as discussed in section 2) are equal and corresponding edges go to sharable nodes.
>
> *Linking*: two entities are linkable if their attributes are equal and corresponding edges go to linkable nodes; or if both are class types with the same tag, one is a definition, and the other is a forward declaration.

The difference between sharing and linking is that linking will combine a class forward declaration with a class definition of the same name, while sharing will not. For example:

```
TU1                TU2                      TU3

struct A;          struct A { int i; };     struct A { int i; };
```

Figures 1, 2, and 3 are the diagrams for these TUs with, respectively, no combination, sharing, and linking.

Figure 1: No combination

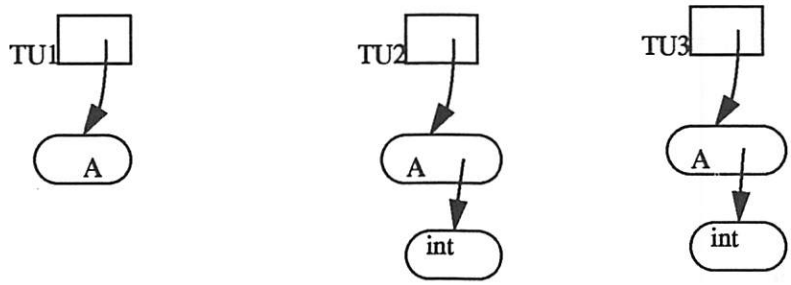


Figure 2: Sharing

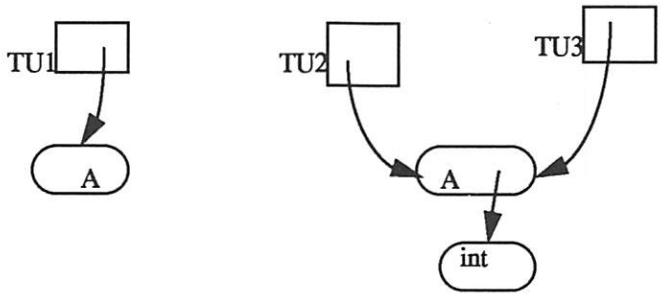
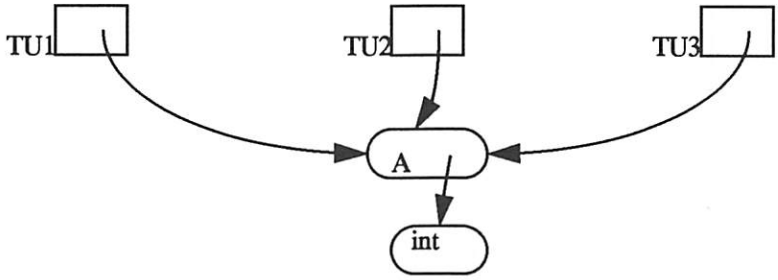


Figure 3: Linking



If we look closer at these definitions of sharing and linking we find two problems. The first problem is that a graph with cycles—for example, a class containing a pointer to itself—can result in infinite regress when one tries to apply these definitions. The corrected definitions are more cumbersome and can be found in the Appendix.

The second problem is that, although we define the relations sharable and linkable, we do not say which sharable types are shared, nor which linkable types are linked; we have not really defined sharing and linking. There is an easy solution for sharable types: share all types that are sharable. But for linkable types this is impossible. For example, consider the following three TUs:

```
TU1                 TU2                 TU3

struct B;           struct B {          struct B {
                          int i;              char c;
                    };                  };
struct C {                              struct C {
     B* bp;                                 B* bp;
};                                      };
```

The first `B` is linkable with the second `B` and the third `B`—but it can only be linked with one of them. Furthermore, the two `C`'s, which are linkable, can only be linked if the first and third `B`'s are linked, not if the first

and second `B`'s are linked.

In our database linking is performed eagerly during TU combination, and a linking decision is never "rethought" after it is made. For example, if the TUs above were added to our database in numerical order, the result would be two defined `B`'s and two defined `C`'s; but if the TUs were added in the order TU1, TU3, TU2, the result would be two defined `B`'s and only one defined `C`.

Note that the state of the database with linking depends on the order in which TUs are added. This is an unfortunate property, because it can make bugs more difficult to reproduce. Sharing does not have this problem.

In the following section we explore situations where there are different class definitions with the same tag.

5. Segregation

In C++, classes come in three varieties:

- *external classes* (their tags have *external linkage*);
- *internal classes* (their tags have *internal linkage*); and
- *local classes* (local to a function; their tags have *no linkage*).

A class in file scope is external if and only if another external entity (ultimately, a variable, function, static data member, or non-inline member function) depends on it. For example:

```
TU1

struct A { int i; };            // This A has external linkage
void f(A);                      // because f depends on it.

TU2

struct A { int i; };            // This A has internal linkage.
void g() {
      struct A { int i; };      // This A has no linkage.
}
```

The fundamental difference between external and internal classes is that C++ requires all definitions of an external class with a given tag to be identical; but there is no such restriction for internal classes.

We probably want to *segregate* these three kinds of types, not to combine one kind with another. For example, although the three definitions of `A` above appear sharable or linkable, if we are segregating according to linkage then they must remain separate nodes.

The variants we measure in section 6 are:

- "No segregation": no distinction is made between internally linked and externally linked classes.
- "Segregation": internally linked and externally linked classes are segregated. The two kinds of classes are combined independently of each other, and can have different combination modes.

In both variants local classes are never combined with each other, nor with internal or external classes. In the programs we measure there are very few local classes.

C++ language issues have an impact on segregation and the combination modes for internal and external classes. Most importantly, this example gives rise to a semantic error unless we segregate:

```
TU1                 TU2                       TU3
struct A;           struct A {                struct A {
                          char c;                   int i;
                    };                        };
                    // internal class
extern A x;                                   extern A x;
```

The A's in TU1 and TU3 are external; the A in TU2 is internal. If these TUs are TU-combined in numerical order, and we do not segregate, then the A's in TU1 and TU2 will be linked, leading to the variable x incorrectly seeming to have two conflicting types. If we segregate, then the right thing happens: the A's in TU1 and TU3 are linked, and there is no problem with the type of x.

Segregation is also necessary if we are to diagnose a violation of the one definition rule (discussed below) for external classes; and segregation is of course necessary if a we want a browser to distinguish internal from external classes.

Clearly we would prefer to segregate. But given that we want to segregate, we must decide how (if at all) to combine each kind of class. We consider external classes, then internal classes; for each, the answer hinges on a C++ language issue.

External classes fall under the *one-definition rule* (*ODR*).[1] The ODR states that all external definitions of a tag must be the same. Since all external definitions of a tag are the same, they, together with all external forward declarations of the tag, can be linked to form a single definition. This implies that we should link external classes.

We may even be tempted to think that we can link external classes using a simplified, easier-to-implement algorithm that does not tolerate differing definitions of the same tag. If we encounter a non-linkable definition, so the argument goes, we can give a fatal error message and reject the TU from the database. Unfortunately, no C++ implementation that we know of enforces the ODR, and so we expect that many programs violate it accidentally. One of our example programs, cxxcomp, violates the ODR deliberately.[2] As a result, a database that aims to handle existing code cannot restrict itself to programs that conform to the ODR, and a linking algorithm must handle differing definitions of the same external tag.

Internal classes might be said to fall under the "many-definition rule". Even if internal class definitions in separate TUs are identical, they still define different types. One can construct a program to verify this using run-time type information (RTTI) [Stroustrup & Lenkov], a set of features recently added to C++ that allow explicit manipulation, and testing for equality, of representations for the types of expressions. In fact RTTI (and exceptions) require a C++ compiler to generate a database of types, called the typeinfo database, to be accessible by each program at run-time. Much of what this paper says applies to the design of a typeinfo database. In the next section, scenario D was included because of its possible use in a typeinfo database.

Two corner cases, tagless classes and nested classes, complete our survey of issues related to segregation.

Tagless classes are a conundrum in C++, since "C++ relies on name equivalence, *not* on structural equivalence of types" [Ellis & Stroustrup]. The ANSI/ISO C++ committee is currently debating the status of tagless classes. Our implementation sidesteps the issue by generating a unique pseudo-tag for each tagless class, then allowing structurally equivalent tagless classes to be combined between TUs in spite of their differing pseudo-tags. Linkage for tagless classes is computed just as for tagged classes.

Linkage for nested classes is also ambiguous in C++. Does external linkage propagate from a class to a class nested within it? From a nested classes to its containing class? For our implementation the answer is "yes" and "no", respectively, but the "no" could be argued with.

---

1. The ODR is found only in implicit form in [Ellis & Stroustrup]. The need for it is generally accepted in the ANSI/ISO C++ committee, but its precise definition is still being debated.
2. The cxxcomp sources violate the ODR as follows: many member function declarations are conditionally compiled out in most, but not all, TUs. This speeds up compilation of the cxxcomp sources.

6. Measurements

We measured three medium-sized C++ program fragments loaded into our database, in each case trying all combinations of the parameters elision, segregation, and combination mode for which we could imagine a use. We call a particular combination of the parameters a *scenario*. A scenario is named with a letter, indicating the segregation and combination modes (see table 1), followed by a "+" or "–" that indicates, respectively, no elision or elision. (The "–" is intended to suggest that elision "subtracts" entities.) For example, the scenario E– is elision, segregation, linking of classes with external linkage, and sharing of classes with internal linkage.

Table 1: Scenarios Measured

Scenario Letter	Segregation	Combination Mode	
		External Classes	Internal Classes
A	Yes	No combination	No combination
B	Yes	Share	Share
C	No	Share	
D	Yes	Link	No combination
E	Yes	Link	Share
F	Yes	Link	Link
G	No	Link	

Scenarios A (that is, A+ and A–) are the "naive database", with no combination.

Skipping ahead for a moment, scenarios G, linking with no segregation, are what the current release of ObjectCenter implements. Scenarios B through F are the alternatives we considered.

Scenarios B and C are sharing with, respectively, segregation and no segregation. Because sharing is much simpler to implement than linking, it is a viable implementation choice. We were interested to see how much larger the database would be with sharing than with linking.

Scenarios D, E, and F all involve linking of external classes. For internal classes they involve no combination, sharing, and linking, respectively. D is the most straightforward implementation for a database that must be completely accurate about types, such as a typeinfo database, since an internal class in one TU has no relationship to an internal class in another TU (even if they look identical). E allows a smaller, but less straightforward implementation of a typeinfo database: each reference to a type must be annotated with an identifier for the TU the type is in. The TU annotation allows two internal classes whose representations are shared to be distinguished. A database with fewer accuracy constraints may choose to implement E without the TU annotation. Finally, scenarios F produce databases even smaller than E, but at the price of a further inaccuracy: the linking of an incomplete internal class in one TU with an arbitrary complete internal class with the same name in some other TU.

The program fragments we measure are shown in tables 2 and 3.

The measurements do not consider the additional libraries normally linked with these programs—for example, the cbrowse measurements do not include the OI class library—but the measurements *do* consider the included header files from those libraries.

The line counts reveal that we have chosen three programs with different organizations. Most of the lines of cxxcomp are in the top-level source files; cbrowse is mostly header files; iv is about half and half. Note that

Table 2: Programs Measured

Name	Header Files	TUs	Description
iv	247	53	Some of the "doc" editor, and some of the InterViews 3.1 library itself.
cbrowse	289	10	Some of the ObjectCenter user interface. This program uses the OI class library [Aitken], which has many large header files.
cxxcomp	80	42	A proprietary C++ compiler.

Table 3: Line Counts on Programs Measured

Program	Raw		Cooked[a]		After pre-processing, cooked
	Header files	Top-level sources	Header files	Top-level sources	
iv	19661	15434	12013	12310	58280
cbrowse	45994	4615	31109	3743	182586
cxxcomp	18723	69627	14877	54434	169890

a. "Cooked" means excluding blank lines and comments.

even for cxxcomp, most of the lines a translator sees after header file expansion (see the "after preprocessing, cooked" column) come from header files.

Two other differences in these programs are important: cxxcomp is a complete program (except for the C++ standard library), while the other two are fragments; and in cbrowse essentially the same (large) set of header files are included in each top-level source, while the other two programs are less uniform in their inclusion.

We measure two quantities in our database. Our first quantity is the *number of defined classes* in the database. We rely on this one measurement because it is a reasonable predictor of the database size, at least for our database; and because it is an abstract number, mostly independent of the details of our database.

It is not surprising that the defined class count is a good predictor of database size: the representation of a defined class in our database is larger than the representation of nearly any other language entity.

Our second quantity is the *size of the C++-related database* in kilobytes. We are measuring the data structures produced by our C++ front end to represent functions (though not their executable code), variables, templates, and types in the database. We do not measure representations of identifiers and macros, nor do we measure executable code and other back end data structures, although these are also in our database.

Figures 5 and 6 plot our two quantities against the fourteen scenarios. Figure 5 plots the number of defined classes (vertical axis) for each scenario, and figure 6 plots the total database size (vertical axis) for each scenario. Low points are better (fewer classes or a smaller database).

Each graph has two sets of points. The first set are for the "+" (no elision) scenarios and are represented by a "+", though this mark often looks like a vertical tic. The second set are for the "–" (elision) scenarios and are represented by a square. The lines connecting the points in a set show trends: a slope from upper left to lower right is an improvement, the steeper the better.

Several features are apparent in these graphs.

Figure 5: scenarios vs. number of defined classes

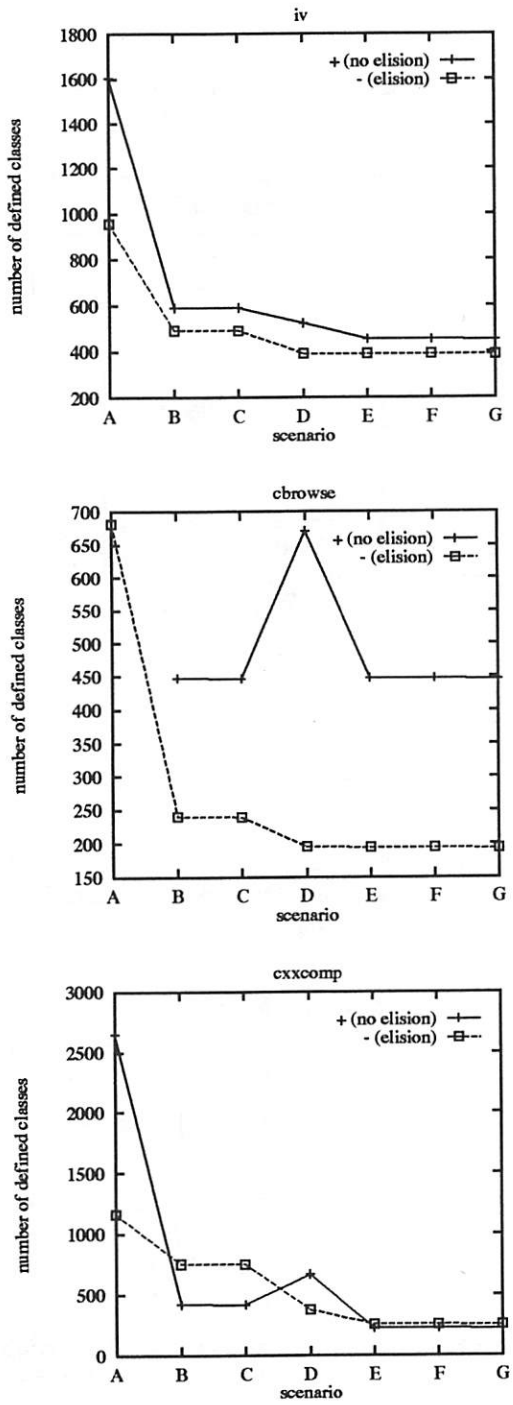
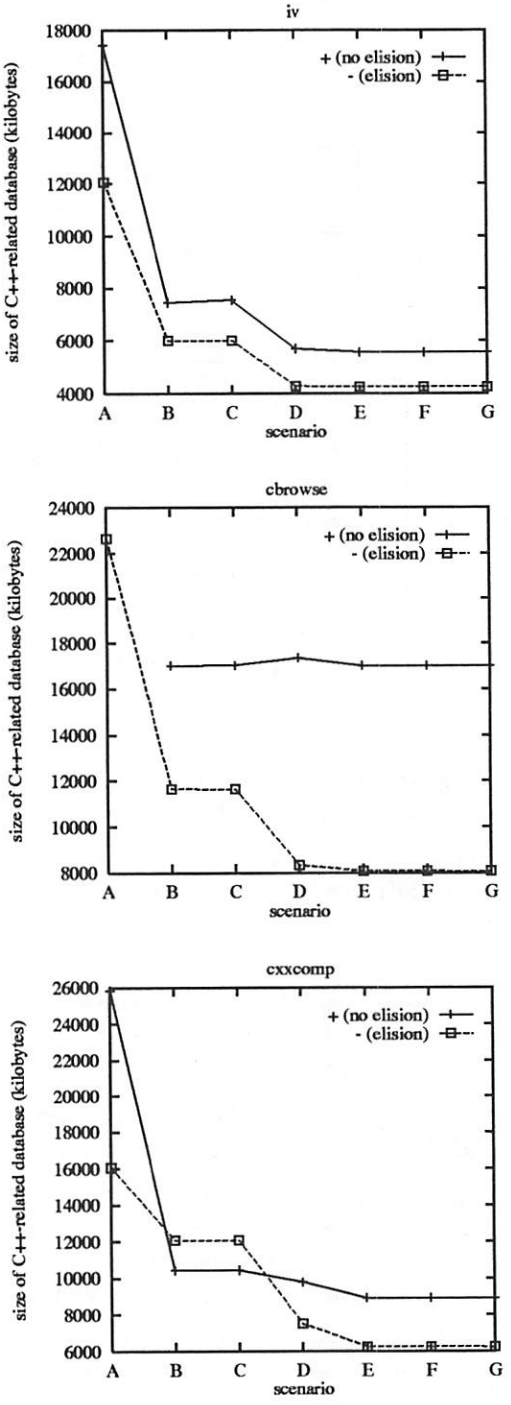


Figure 6: scenarios vs. size of C++-related database (KB)

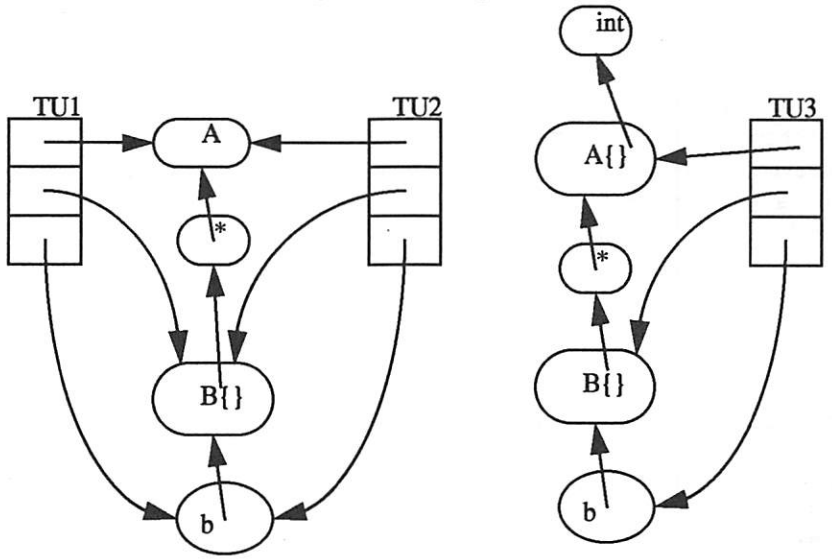


- The improvement is dramatic between no combination and sharing (scenarios A vs. B).
- Between sharing and linking (scenarios B vs. F, or C vs. G), the improvement is less dramatic but still significant. The following example suggests why this is so:

```
TU1                 TU2                 TU3
struct A;           struct A;           struct A {
                                              int i;
                                        };
struct B {          struct B {          struct B {
     A* p;               A* p;               A* p;
};                  };                  };
extern B b;         extern B b;         extern B b;
```

No combination produces three nodes for `B`. Linking combines them all and produces one node for `B`. Sharing (scenario B+ for this example) produces two nodes for `B`; it is diagrammed in figure 7. Because

Figure 7: Sharing



`A` cannot be shared between TU3 and the other TUs, `B` (which depends on `A`) cannot be shared either. We call this a *cascading combination failure*. In actual programs classes are often heavily interdependent; as a result, when one class fails to share (or link) the cascade can be many tens of classes.

- There is no data for the naive (A+) scenario of cbrowse, because that scenario exceeded a normally reasonable fixed limit in our database.
- Again for cbrowse, in the absence of elision there was no improvement from sharing to linking (B+ vs. F+). This is because the TUs of cbrowse are homogeneous; they all include the same header files, and so they all declare the same classes in the same way. Without some difference between TUs, sharing does not fail.
- Elision is uniformly better than no elision for no-combination (scenario A) and for linking (scenarios E, F, and G).[1] This is not surprising, since elision deletes classes.

1. A minor bug in our implementation of elision makes elision look very slightly worse than no-elision for cxxcomp scenarios E, F, and G in "number of defined classes". In reality elision should look very slightly better for those scenarios. No other result was significantly affected by this bug.

- Elision sometimes does not interact well with sharing. This is evident in cxxcomp scenarios B and C. The reason is that elision can demote a class definition to a forward declaration in one TU but not others, creating an opportunity for cascading combination failure. In general, sharing prefers homogeneity, and elision destroys homogeneity. One might expect to observe this interaction with cbrowse, since its TUs are so homogeneous; but so many defined classes go unused in cbrowse that their removal by elision (which reduces the number of classes) overwhelms the bad interaction (which increases the number of classes).
- For cxxcomp scenarios E, F, and G, "number of defined classes" appears not to favor elision over no elision. Yet elision is sharply favored in "size of C++-related database". This disparity seems to be due to elision's effect on entities other than classes, notably function and variable declarations.
- The improvement (slope) is steeper between scenarios C- and D- in plotting database size than in plotting number of defined classes. We believe this is due to a subtle effect that we have also observed independently: cascading combination failure hits large, complicated classes more often than small, simple classes. As a result sharing (which has many cascading combination failures) produces a larger database than the number of defined classes would lead one to believe; linking (which has far fewer cascading combination failures) does not.
- In these programs, at least, internal and local classes are insignificant. In none of our programs are there more than two local classes or enumerations. As for internally linked types, they seem significant only in scenario D+ for cbrowse and cxxcomp, where their numbers are inflated by the lack of elision and combination. But even in those scenarios the inflation is insignificant when we consider the size of the database. This disparity between number of defined classes and database size is because most internal classes are simple, and so have small representations.

7. Related Work

7.1 Sharing at Header File Granularity

Most modern programming languages comparable to C++—for example, Ada and the Modula family—are organized around modules. An "import" construct takes the place of `#include`. An imported module, unlike an included header file, has the same meaning in all contexts. As a result, modules provide a natural structuring unit for a program database. For C++, header files can be used as such a unit, but the database must be prepared to see multiple meanings for one header file.

The combination we have talked about so far has all been at *fine granularity*; that is, individual variables, types, etc. are combined (or not). Some databases, however, share at *header file granularity*: all the declarations in a header file are shared (or not) as a unit. These include some versions of the Sun C debugging information database [Linton] [Menapace], the DEC C++ debugging information database [Hamby], the Sun C++ source browser database [Pelegri-Llopart], and the Rational C++ Analyzer [Wilcox] [Wilcox94], part of Rational Rose/C++ product.

Elision essentially prevents sharing at header file granularity, for a different set of entities will likely be elided from a given header file in each TU that includes it.

Since a difference in any one declaration in a header file prevents sharing, great attention must be paid to minimizing such differences. Following is a list of other items that prevent sharing at header file granularity. Most of these factors are inherent in the C and C++ languages.

- Conditional compilation. Here are two common examples, the first surrounding the other:

```
#ifndef UNIQUE_TO_THIS_HEADER
#define UNIQUE_TO_THIS_HEADER

#if DEBUG
/* Declarations... */
```

```
#endif
#endif /* UNIQUE_TO_THIS_HEADER */
```

These very common idioms often cause one instance of a header file to contain a different sequence of declarations than another.

- Redundant declarations. Many translators do not emit a record on any but the first of a sequence of redundant declarations. If several header files contain a declaration `struct A;` and those header files are included in various orders, some inclusions of a given header will seem to declare `A` and some will not.
- Implicit declarations, e.g.:

```
struct A *p;        // Implicit declaration of A.
```

  Most translators do not emit a record on an implicit declaration unless it is the first declaration. This reduces sharing for the same reason that redundant declarations do.
- Automatic generation of source code. Some language processors generate C or C++ source code. These processors often do not emit header files at all, thus defeating any header file-based sharing scheme.
- Static variables. If the description of a static variable includes its address (as in some debugging information formats), then no two inclusions of a header file will be sharable.
- Inline functions. When bodies are generated for these, as is common, they can cause the same sharing problem as static variables.
- Generated names for unnamed types, anonymous unions (which are not types), and function parameters. Some C++ compilers generate a unique identifier for an unnamed entity. A naive name-generation algorithm will generate different names for the same entity in different TUs.

The last two problems are unique to C++. The others affect both C and C++. Nevertheless sharing at header file granularity has proved useful for many databases.

7.2 Database of Declarations

Our database is oriented around such entities as types, variables, and functions—the entities that declarations declare. It is these entities we have elided, combined, and measured. An alternative organization taken by many databases, particularly databases intended to support browsing, is to store only the declarations. Such an organization is very flexible, and it is more amenable to sharing.

One way to categorize program databases is how close they are to the program source, or to the machine code (figure 8). Sharing is easier when a database is closer to source code. For example, in a database of

Figure 8: Spectrum of C++ Program Databases

source code	tokenized source code	database of declarations	our database	object code

tokenized source code sharing is trivial: just share the tokenized representations of header files. On the other end, object code is almost never shared between TUs.[1] In the middle, we have databases of declarations, for which sharing is relatively easy. For our database, as we have seen, sharing is possible but not optimal; we are forced to introduce linking, which is more complicated, to save additional space.

1. As an example of sharing object code between TUs, a linker might share function bodies that happen to be identical. Such redundant functions can be generated by template instantiation.

7.3 Multi-Program Database

For the most part we have assumed that a database contains at most one program. But some databases, such as the Sun Source Browser database, can contain many programs in one database. The essential restriction in designing such a database is that you may share entities, but you should probably not link them. The database may contain any number of definitions for an entity E; to link a forward declaration for E, one would have to pick one of those definitions. Any choice is semantically meaningless or even incorrect; the chosen definition may be from an entirely different program.

This is the same problem that confronted us for classes with internal linkage, but in a multi-program database this problem affects all named entities. Namespaces (a new C++ feature) may ameliorate this problem for newly written code.

7.4 Lazy Loading

Gdb, the GNU debugger, does not combine or elide. Instead, it loads debugging information from object file(s) into its in-memory database in a lazy fashion: the debugging information for a TU is only loaded when absolutely necessary. This trick is usually portrayed as a time-saver, but it is also a space-saver.

Gdb would probably benefit from combination, which is orthogonal to lazy loading. Elision is not possible, since there is not sufficient dependency information in object files.

7.5 Ruthless Simplification

Cia and cia++ are designed to handle large programs. We believe that the design achieves its scalability not so much by tricks of the kind we have presented—although both cia and cia++ seem to implement linking—as by ruthless simplification. They store information only about global entities and the relationships between them; the rationalization is that these relationships are the most important in examining the structure of large programs.

The opposite approach is that taken by Reprise [Rosenblum & Wolf], the Rational C++ Analyzer, and Alf [Murray]. These AST (abstract syntax tree) databases store nearly everything about the program. One of them, Reprise, has no combination between TUs. Rational has a particularly radical form of sharing at header file granularity, in which header files can be user-specified to behave as though they were importable modules. [Murray] hints that Alf will also prefer header files that have the same meaning in all contexts.

7.6 Multiple C++ Translators

One final point we feel is crucial, although it is not reflected in our measurements. In these examples, our database has only one C++ translator generating graphs. This is true for most other databases. Were a database to have multiple, independently implemented translators generating graphs, then combination and enforcement of the ODR would become problematic.

Because there are multiple implementors each interpreting an inevitably ambiguous specification of what output the database expects, small details in the representation of programs will differ, and any difference is enough to prevent sharing or linking. We have observed these differences in the various C compilers that generate STABS debugging information. Simple details that tend to differ are such things as representation of classes and enumerations with no members; and the distinction between types with identical machine lengths and formats, such as (on many systems) `char` and `signed char`, or `double` and `long double`.

Given that a C++ program database can be rougly as complicated as the language itself, it is difficult to imagine that these problems can be completely avoided. The database that will face these problems in the near future is the typeinfo database; luckily, the requirements on this database are simple enough that it should be possible to give it an unambiguous representation specification.

8. Conclusion

We state our conclusions as advice to designers of program databases on how to save space.

Our first set of advice is aimed at those designing databases with constraints similar to ours: no more than one representation for a given program entity in a given TU; combination must generate data structures designed for no combination, or for handling no more than a single TU. These databases might include a debugging information database (on disk or in memory) or a run-time typeinfo database.

- Segregation of internal from external classes is necessary for correctness, and imposes no significant space penalty.
- Most classes are external, so how external classes are combined is important. Future releases of ObjectCenter will probably continue to link external classes and to elide by default.
- The other reasonable alternative for external classes is sharing. In this case elision should probably not be available or at least should not be the default, since it does not mix well with sharing. This alternative is much easier to implement.
- Internal classes may be not-combined, shared, or linked. No-combination preserves program semantics most rigorously, but this kind of correctness is probably only important to a typeinfo database. Sharing preserves semantics except for type identity, and is the mode future ObjectCenter implementations will probably use for internal classes. Linking gives false definitions to undefined internal classes.

Our second set of advice is more general, aimed at those designing C++ program databases. We have no definite advice, but we can pass on strategies used in other databases:

- Sharing or linking of classes will almost certainly save space and time.
- Ruthlessly simplify and limit the information in the database, keeping only what is necessary for your purposes. This strategy, we believe, enables the cia++ database to handle large C++ programs.
- Represent only declarations, not declared entities. Sharing becomes easier and more profitable. This is the strategy used in browsing databases such as the Sun Source Browser and the Rational C++ Analyzer. If you are sharing at header file granularity, do not elide unused declarations.
- If your database is built by extracting data from another database, extract lazily. This strategy is used by gdb.
- If there are multiple, independently implemented translators feeding into your database, combination will probably suffer due to slight variations in output between the translators. Combat this problem by ensuring that the output of the translators is rigorously specified.

It is an interesting question whether an implementation should diagnose a violation of the one-definition-rule for classes with external linkage. It is clear that a debugging environment such as ObjectCenter should. However, we conjecture that many existing programs will violate this rule; it is important to allow them to execute.

Acknowledgments

We would like to thank those the many patient people who answered questions from the authors, particularly Geoff Wyant, Judy Grass, Robin Chen, Tom Wilcox, Ed Pellegri-Llopart, and David Chase.

Thanks are due to Geoff Wyant, Ann Wollrath, and Webb Stacy for their helpful comments on drafts of this paper. Particular thanks are due to Jim Waldo, whose suggestions and detailed comments on drafts were timely and very insightful.

Of course, any mistakes are the fault of the authors or of the computers they used to prepare the paper.

Trademarks

SPARCworks is a trademark of SPARC International, Inc., and is licensed exclusively to Sun Microsystems, Inc. ObjectCenter is a trademark of CenterLine Software, Inc. Rational and Rose/C++ are trademarks of Rational. Other product or service names mentioned herein are trademarks of their respective owners.

References

[Aitken] G. Aitken, "OI: A Model Extensible C++ Toolkit for the X Window System," *Proc. 4th Annual X Technical Conference*, January 1990.

[Chen] Y.-F. Chen, "The C Program Database and Its Applications", *Proc. USENIX Summer '89*, pp. 157-171.

[Chen94] Y.-F. Chen, personal communication, 1994.

[Ellis & Stroustrup] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Grass & Chen] J. E. Grass and Y.-F. Chen, "The C++ Information Abstractor", *Proc. USENIX C++ '90*, 1990, pp. 265-275.

[Grass] J.E. Grass, personal communication, 1994.

[Hamby] J. Hamby, personal communication, c. 1992.

[Linton] M. Linton, "The Evolution of Dbx", *Proc. USENIX Summer '90*, June 1990, pp. 211-220.

[Menapace] J. Menapace, J. Kingdon, and D. MacKenzie, "The 'stabs' debugging information format". In file gdb-4.12/gdb/doc/stabs.texinfo in the distribution of GDB version 4.12, available from numerous ftp servers such as prep.ai.mit.edu.

[Murray] R. B. Murray, "A Statically Typed Abstract Representation for C++ Programs", *Proc. USENIX C++ '92*, 1992, pp. 83-97.

[Pelegri-Llopart] E. Pelegri-Llopart, personal communication, 1994.

[Rosenblum & Wolf] D. Rosenblum and A. Wolf, "Representing Semantically Analyzed C++ Code with Reprise", *Proc. USENIX C++ '91*, 1991, pp. 119-134.

[Rovner] P. Rovner, *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language*. Xerox PARC technical report CSL-84-7, July 1984.

[Stroustrup & Lenkov] B. Stroustrup and D. Lenkov, "Run-Time Type Identification for C++ (Revised yet again)", doc. no. X3J16/92-0121 WG21/N0198, ANSI/ISO C++ Committee Mailing, 1992.

[Wilcox] T. R. Wilcox, USENIX C++ Advanced Topics Workshop presentation, 1992.

[Wilcox94] T. R. Wilcox, personal communication, 1994.

[Wyant] G. Wyant, J. Sherman, D. Reed, and S. Chapman, "An Object-Oriented Program Development Environment for C++", *Conf. Proc. of C++ At Work 1991*.

Appendix: Corrected Definitions of Sharable and Linkable

The corrected definition of sharable is as follows:

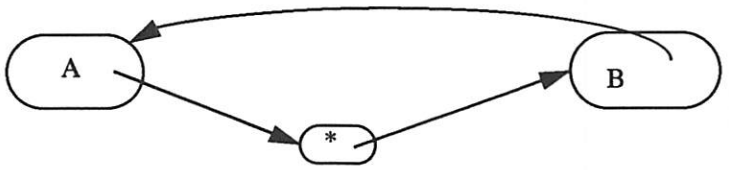
> Two types T and U are *sharable* if every pair of paths of equal, finite length, beginning at T and U and proceeding at each step along corresponding edges, terminate at entities whose attributes are equal, and which possess only edges corresponding to edges in the other.

The definition of linkable is corrected in a similar fashion. Note that paths of zero length are part of the definitions, and are necessary for types that are *not* involved in a cycle.

For example, two translation units containing

```
struct B;
struct A { B* p; };
struct B : A {};
```

result in two copies of the following type graph:



When we try to apply the original definition of sharable to the A's, we find that the definition of whether the A's are sharable depends on itself. When we apply the corrected definition, we find that although we have to examine an infinite number of (pairs of) paths:

> [A], [A, B*], [A, B*, B], [A, B*, B, A], ...

the two A's are indeed shareable.

The infinity can be removed, though doing so is not necessary and makes the corrected definitions even more cumbersome.

A Dossier Driven Persistent Objects Facility

Robert Mecklenburg, Charles Clark, Gary Lindstrom and Benny Yih

University of Utah Center for Software Science
Department of Computer Science
Salt Lake City, UT 84112

E-mail: {mecklen,clark,gary,yih}@cs.utah.edu

Abstract

We describe the design and implementation of a persistent object storage facility based on a dossier driven approach. Objects are characterized by dossiers which describe both their language defined and "extra-linguistic" properties. These dossiers are generated by a C++ preprocessor in concert with an augmented, but completely C++ compatible, class description language. The design places very few burdens on the application programmer and can be used without altering the data member layout of application objects or inheriting from special classes. The storage format is kept simple to allow the use of a variety of data storage backends. In addition, these dossiers can be used to implement (or augment) a run-time typing facility compatible with the proposed ANSI C++ standard. Finally, by providing a generic object to byte stream conversion the persistent object facility can also be used in conjunction with an interprocess communication facility to provide object-level communication between processes.[1]

1 Motivation

The basic problem of a persistent object store (POS) is simply stated:

> Given a reference to the root node of a graph of objects generate a data stream which can be used to reconstitute the original object graph at a later time.

Many approaches have been pursued to solve this basic problem (see Section 12 for a summary). The utility of these approaches is governed by the constraints they impose on application code in such dimensions as (i) language or compiler extensions, (ii) mandatory inheritance from library base classes, (iii) system transformation of application source code, (iv) expansion of object size, (v) mandatory presence of virtual function tables, and (vi) programmer declaration of supporting functions and observance of programming style restrictions.

We describe a new approach which poses no constraints in (i) – (v), and minor client obligations in (vi). Our approach is based on preprocessor-generated dossier objects[14], which drive fully polymorphic (i.e., applicable to all types) load and store functions. In addition to supporting object persistence, our approach provides a fully general means for transporting object graphs in address space independent form (i.e., "pickled", with "unswizzled" pointers). Our design has been motivated by the stringent demands of a large (750,000 line) C++ CAD/CAM/visualization application[2].

[1]This research was sponsored in part by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of ARPA, or the U.S. Government.

2 What Is An Object?

We begin by defining our unit of persistence, which we term an *object*. While some approaches take this to be C++ class instances, this basis is too narrow for applications such as our CAD client, which make extensive use of graphs of vectors and structures, with semantically significant sharing relationships. Hence we define an *object* to be a contiguous region of memory whose type is known either through

- static type information,
- dynamic type information (e.g., virtual function table), or
- information provided by the application programmer.

An object is identified in an application by a pointer or reference to its first address along with some notion of its bounds (derived from type information). We explicitly disallow pointers to the interior of objects. An *object graph* consists of a collection of objects formed into an arbitrary graph by pointers embedded in the objects. An object is identified in the persistent store by a unique *object identifier* (OID). An application requests objects by OID and can access the OID of an object given its virtual address in the application.

One consequence of this definition is that data members of objects cannot be read or written independently of the containing object. This might occur when a data member is passed as an argument to a function, then saved. Given that we know the type and size of the data member this restriction might be lifted, however, we have not yet had occasion to do so.

3 Client Constraints

To be as convenient as possible a POS must minimize the impact of its use on application source code and the software development process while at the same time maximizing functionality. Among the features of a POS, we feel the following to be important: minimal impact on object layout and class declarations, allow the use of standard language tools, provide object access from a variety of hardware platforms, provide object access after class mutation. We discuss each of these requirements in turn.

The POS should not require "large" changes to class definitions. In particular, any system which requires altering the class layout by adding data members, virtual functions (where none existed before) or additional base classes is unacceptable. Such a system would impose storage overhead and incompatibilities intolerable to many applications. However, adding virtual functions to a class with an existing virtual function table is an acceptable change which would allow more convenient use of the storage facility. If this modification were allowed (but optional) it would provide a convenient interface for application specific classes while still allowing library classes (for which there is no source code) to persist.

One of the biggest problems caused by many POSs is the requirement for non-standard language tools (e.g., special compilers) to enable objects to persist. These tools either parse an extended language syntax (translating into standard C++), generate augmented class implementations, or some mixture of the two. Our group, having worked on large software projects using these approaches and finding them burdensome, chose to require the class definition be written in standard C++. This means that there is only one class definition (with no additional semantic information in other files) and that applications can be compiled and run (albeit without persistence) with or without the persistent objects facility. This significantly simplifies porting, piece-wise development and testing of applications.

Once a POS is integrated into an application or organization its use quickly becomes fundamental to the project and the persistent objects themselves become a valuable resource. As such,

it is often unacceptable to abandon the database when new hardware or software is acquired or when class definitions change. Furthermore, as the size of the database grows evolving the data *en masse* becomes a significant burden. We feel a more reasonable approach is to integrate platform heterogeneity and type evolution cleanly into the persistent store allowing for lazy transformation of objects to the reader's requirements.

We discuss other, less major, constraints on the POS as they arise.

4 An Object Description Language

We now address the need for a language in which to describe objects. An object which is an instance of a primitive C++ type may be described simply by its standard type name. One may reasonably expect that an object which is an instance of a class may be described by the C++ declaration of that class. Indeed, to a first approximation, that is correct. Unfortunately, there are several "extra-linguistic" patterns of use which are not sufficiently described by standard C++ syntax, particularly with respect to dynamically sized objects (e.g., strings and other vectors). The problem is to identify important idioms required by applications and to provide an annotation mechanism which does not invalidate the use of standard language tools. In addition to these annotations, the POS may require classes to provide various semantic handles to allow storage and retrieval.

The most important idiom in C++ which is not adequately described by class declarations is the use of pointers to access dynamically sized regions of memory. Strictly interpreted, the declaration:

```
char *cp;
```

identifies a pointer to an unknown number of characters. By convention the number of characters is determined by a *sentinel value*, in this case the null character. The sentinel value technique for dynamically sized data can be used with any data type, but is most typically used with pointers and integral types where the zero bit pattern is the most common sentinel. A competing style for identifying the size of dynamically sized memory regions relies on a pair of data values:

```
int   n;    // size of cp
char *cp;
```

where the dynamic size is stored explicitly in a separate data member.

Static data members of a class pose a different sort of problem for a POS. Indeed, one may question whether static data members should persist at all. Often these data members are used to resolve issues inherent in run-time data management. For instance, an application might maintain an *extent* list of all allocated instances. Such a list acquires a completely different meaning in a persistent store owing to the shared, distributed, and concurrent nature of the store. Our approach is to allow the application programmer to indicate whether static data members should persist. However, we chose not to manage concurrent access. Aside from ensuring consistent concurrent writes for single data members we do not assume any further capabilities of the underlying POS such as notifying readers of updates to shared data. Similar to static data members there may be non-static data members which the programmer does not want saved. For example, an object might contain a pointer to a buffered file structure which has no meaning (or a different meaning) when stored in a POS. These nodes can be annotated as *orphaned* objects; their value will not be stored and their pointers will not be traversed.

Unions present an interesting problem due to the ambiguous nature of the type information available. In particular, if a union contains both a pointer and a non-pointer, should the pointer be traversed? The current approach is to require unions to be enclosed in a class containing at least the union and an integer which is used as the union discriminator. We feel this is a reasonable

compromise between a completely arbitrary decision (on our part) and completely user defined behavior (which we have no way of specifying).

Finally, pointers to member functions are not yet supported. Since the implementation of pointers to member functions does not require a virtual memory address, it may be easy to save and restore them. Unfortunately, the possible variance in their implementation makes any one storage technique non-portable. It may be possible to encode the pointer to member function implementation along with the pointer itself in the persistent object, but we have not investigated this technique.

4.1 Syntactic Considerations

How can these annotations be applied to a class definition if standard compilers are used and no additional files are consulted? There are three basic approaches: parameterized classes, embedded annotations in comments, or augmented identifier names. These would be used to identify the three basic annotations discussed above:

- dynamically sized arrays terminated by a sentinel,
- dynamically sized arrays whose size is defined by an associated integer, and
- objects which are to be omitted from the persistent store (orphaned).

Templates could be used to identify data members with these attributes by defining a template class for each of the annotations. For instance, a simplified template for a null terminated array might be:

```
template <class T>
class null_terminated {
    T *p;
public:
    operator T*() { return p; }
};
```

This would then be used in an application by replacing a simple pointer declaration with:

```
class X {
    null_terminated<char>   cp;
    ...
```

Unfortunately, this approach introduces all the problems associated with a smart pointer class[9]. In particular, our template class does not interact well with the `const` keyword, nor is it guaranteed to have equivalent performance characteristics. Applying this class to existing programs would entail significant source code (and possibly algorithmic) changes. Although we feel this is a syntactically elegant solution to the annotation problem, it is only useful in a restricted domain (e.g., writing new applications).

The second approach to annotating classes places comments adjacent to data members containing keywords identifying various attributes. Similarly, the third approach uses the data member name itself (or its type name) to contain the attribute. An example of the later is:

```
typedef char char__null; // Null terminated string.
char__null *path;

typedef int int__sized; // Integer sized string.
int__sized n;
char *     name;
```

```
// reconstructor_t - Type used to identify the reconstructor.
enum reconstructor_t { RECONSTRUCT };

// char__null - Annotated type for a null terminated array of char.
typedef char char__null;

// int__sized - Annotated type for an integer sized array.
typedef int int__sized;

// dictionary_c - A simple association table.
class dictionary_c {
 public:
    dictionary_c( reconstructor_t );
    ~dictionary_c();
    ...
    dossier_c *         __get_dossier() const;
    void                __load_store_hook( int when );

 private:
    char__null *        name_;  // The dictionary name.
    int__sized          len_;   // The size of the table.
    dict_elem_c *       table_; // The association table.
};
```

Figure 1: A typical class with annotations.

We implemented this last technique for several reasons. We consider the annotations themselves to be an essential part of the type information which a programmer must usually omit due to limitations in C++ declaration syntax. By augmenting the type declaration we are making this type information explicit at the appropriate time and place. Also, it does not interfere with a standard commenting style for class declarations. Finally, it allows us to experiment with a novel annotation technique which we have not seen used before. Annotating the type of the data member (rather than the member itself) leaves the application programmer free to select meaningful member names unencumbered by the annotations. The currently supported annotations are:

`__null`	dynamically sized, zero terminated
`__sized`	dynamically sized, this member is the size, following member is the pointer
`__orph`	an orphaned object, don't save

Figure 1 shows a simple dictionary class augmented with several annotations.

4.2 Application Object Services

To recreate the original semantics of a persistent object the POS must be able to request certain services of the object. Most importantly, that of object allocation and creation. Likewise, the object may require that the persistent store relinquish program control to the object at special times, often just prior to storage and just after loading.

Often the implementations of objects have highly specific meanings associated with the application or environment which do not persist well. Examples of such problems include storing hash tables and file handles. As with other members the writer of the object must annotate the stored instance with information allowing the reader to reconstitute a similar object with semantics equivalent to the original object. For a hash table, the reader may have a different hash function or table

size and therefore must rehash the members of the table. For a file handle, the reader must find and open the file and set the current position. An annotation on a declaration cannot transmit this information (and indeed, may not have the information to transmit). To allow for this type of application specific behavior the programmer can define load and store *hooks* which are called by the POS during object I/O. The load/store hook has a special name and type signature recognized by the dossier generator:

```
void __load_store_hook( int when );
```

This member function is added to the class declaration of any class requiring special handling during I/O. The function can be called under three circumstances (indicated by the `when` parameter): after loading an object, before storing an object, and after storing an object. Figure 2 shows a typical load/store hook for those classes requiring one.

When an object is restored from the POS several application and implementation specific initializations must be performed. The most obvious of these is setting the virtual function table pointer. This can be done in a variety of ways: from using the `new` placement syntax and having the application programmer invoke the constructor to copying the pointer from an initialized sample instance. The later approach does not allow for the application to gain control during object allocation and is therefore unacceptable. Using the `new` placement syntax has the problem of compatibility with other software packages (including the application's classes). A compromise requires the application class to define a special constructor which we call the *reconstructor*. This approach allows classes to overload `new` and `delete` and to gain control during object construction. The reconstructor is identified by its type signature:

```
<class_name>( reconstructor_t );
```

In fact, the reconstructor can be omitted if there exists a default constructor which performs the same function. That is, the default constructor does not have any unwanted side effects and does not assume that the initial values of the object will be seen by the client application code.

Figure 2 shows the typical implementation of a reconstructor. Reconstructors usually have no body since their only duties are to invoke the class (or application) specific memory allocator and to set the virtual function table pointer(s). The actual data members will be overwritten with values from the loaded persistent object.

Finally, to allow convenient use of the POS with polymorphic objects we encourage the application programmer to declare a virtual function for accessing the dossier of a class:

```
virtual dossier_c *__get_dossier() const;
```

This allows the application and POS interface to access the dossier of conforming objects simply. For objects which do not support the `__get_dossier` member function, the application must provide the dossier handle explicitly. This is done by calling a dossier lookup function which accepts the string name of the requested class and returns a pointer to the dossier. These interfaces allow simple and convenient access for classes under application programmer control, while still allowing other classes to persist. After the dossier for the root object is obtained, dossiers for other objects in the graph can be accessed through the root object dossier.

Once an application's class declarations (e.g., `.h` files) have been adapted to express these extra-linguistic features, they become the application's class description. These files are read and analyzed by a preprocessor based on the C++ grammar written by James Roskind[23]. The preprocessor emits auxiliary C++ files which construct instances of class dossiers embodying the class descriptions, including associated annotations. These emitted files are compiled and linked, along with a support library, into an application to implement the client side of the POS. Note that client source files are only read, not transformed, in this process. The application causes an object to persist through an explicit `store` function call. Similarly, objects are loaded from the persistent store by calling a `load` function with the appropriate OID.

```
dictionary_c::dictionary_c( reconstructor_t )
{
}

void dictionary_c::__load_store_hook( int when )
{
    switch ( when )
    {
    case 0: // After loading.
        // Resort the table using current criteria.
        sort_table();
        break;
    case 1: // Before storing.
        break;
    case 2: // After storing.
        break;
    }
}
```

Figure 2: A typical reconstructor and load/store hook.

5 Capture of Compiler and Platform Characteristics

To build a complete description of objects, including data member layout, the dossier generator must mirror the algorithms of the current compiler and would therefore not be particularly portable. We avoid this problem by separating the dossier into machine/compiler independent and dependent portions. The compiler independent portion is constructed by the dossier generator while the dependent portion is computed at run-time from auto-configuring code written into the dossier initializer. The compiler and machine dependent structures gather three types of information: size and format of data types, location of data members in objects, and handles on member functions. We discuss each briefly.

To allow dossier code to read and write objects on differing platforms (both hardware and software) the polymorphic I/O code must know the size of each data type and its format when written to a persistent store. Size information is easily acquired through the use of the `sizeof` compiler directive. Also, byte order and floating point format must be determined. In the worst case, these characteristics must be explicitly specified for each platform making the dossier source code non-portable. In the normal case, however, byte order can be determined through simple calculations and floating point format can be acquired through host configuration files.

The location of data members and base classes for an object are determined using a technique similar to the ANSI C `offsetof` macro. For each (non-static) data member, its location is determined by taking its address and subtracting the object's base address. This requires that the dossier initializer be either a friend or member function of the class. Base class offsets are calculated similarly by casting a "pointer to derived class" to a "pointer to base class". For example, if class `D` derives from class `B`, the expression:

```
((B *)((D *)8)) - 8
```

returns the offset of a `B` instance within a `D` instance. (The use of a non-zero base address subverts optimizations in various compilers.) This expression is portable across all platforms (that we are aware of)[10].

Finally, the polymorphic I/O operations must invoke class reconstructors and load/store hooks to perform their functions. Since the address of a constructor cannot be computed, we wrap the

reconstructor in a simple C++ function and store its address in the dossier. For uniformity we use the same technique to store the load/store hook in the dossier.

6 The Storage Algorithm

The basic storage algorithm is a simple graph traversal driven by the graph's root object and the dossiers. We begin by retrieving the OID of the object to be saved. If the object does not have an OID, allocate one. Next place the object and its OID into the queue of objects waiting to be processed. The rest of the algorithm proceeds as follows:

Algorithm 1

```
dequeue the next node to process
if the node is unsaved
    run the pre-store hook
    mark the object as saved
    enqueue all embedded pointers (allocate OIDs, if necessary)
    store the dossier, if necessary
    store the object and dossier OIDs, and machine id
    store the object
    store the OID of the target of every embedded pointer
    run the post-store hook
```

Dossiers are just objects so they are stored, along with the objects they describe, using the same algorithm. Of course, only one copy of the same dossier is stored and that dossier is referenced by all instances of that class through its OID. Since a dossier is an object, to be read and written it must have a descriptor, or *meta-dossier*. This meta-dossier is a permanent component in the support library and is never written to or read from a POS or communication channel. The meta-dossier is generated by running the dossier generator over its own data structures.

The storage format is designed to be "retargetable" to different object storage engines and is therefore a mix of low-level formats and high-level information. The storage engines currently in use are a transactional DBM and a simple Unix file interface (an Exodus interface is planned). Writing is performed in the simplest possible way, by copying the machine representation of each data member value to the POS. It is the responsibility of the reader to decipher the writer's format. Since objects are often read and written on a single platform this proves reasonably efficient for local communication and temporary storage.

Retrieving object graphs is similar. The retrieval is initiated by the application with the OID of the root node of an object graph. This node is entered into a queue of nodes yet to be read and proceeds as follows:

Algorithm 2

```
dequeue the next node to process
if the node is not yet read
    load the dossier of the object
    load the binary image of the object
    invoke the reconstructor to allocate memory for the object
    record the new object's address and OID
    copy the values of data members from the binary image to the new object
    for each pointer member set the new address, if available
        if not available, place pointer member on patch queue
    run the post-load hook
else
    return the address of the object
traverse patch queue, setting remaining pointer members
```

The object is loaded as a set of binary values from the original object. The dossier is used to pick through this bag of bits to identify data members and their values. The new values for pointers are accessed by the OID of the target object. Due to cyclic graph structures some objects will not have been read yet, so pointers to these objects must be queued until the desired object has been read.

7 Heterogeneity

Heterogeneity is handled by providing a machine description object which contains information concerning hardware and compiler specific data. In Algorithm 1 a machine identifier is stored along with the OIDs of the object and its dossier. This machine identifier references a structure describing the hardware characteristics (e.g., byte order, floating point format) and software characteristics (e.g., member layout) of the writer. When the data for an object is copied from the binary image of the writer to the run-time memory allocated for the reader machine dependent translations are performed.

Although the translations from one hardware platform to another must be hand-crafted, the actual process of converting values from one format to the other is controlled through the dossiers. To avoid writing n^2 conversion routines a standard intermediate format can be used to reduce the number of conversion routines to $2n$.

8 Object Evolution

Invariably, the classes for objects stored in the POS will change due to changes in the user's requirements and added functionality. It is important that old data continue to be accessible to current applications. There are three basic approaches to evolving an object instance from one class declaration to another:

1. provide accessor functions,
2. copy using a "static" algorithm,
3. copy using a "dynamic" algorithm.

The first technique requires that an application be enhanced with accessors that know the old and new type and offset of the desired data member. This accessor is invoked on the old object and returns a value as if from a new object. This is unsuitable for many applications due to its highly hand-crafted nature. The second technique uses the dossier of the old and new objects to copy data member values one by one from the old to the new object using some fixed algorithm. Types that have changed may be converted if the conversion is sufficiently simple (e.g., int to float) and discarded otherwise (assuming that the old value has no translation). New data members may be initialized to some default value (e.g., zero). Experience with one large project indicates that this is a useful evolution technique for many simple object transformations[16]. Nevertheless, it is insufficient as the only (or even primary) type evolution mechanism. The final technique allows the application programmer to provide a function to translate an object from one version of a class to another.

Dossiers can be annotated with version information and can record translation functions capable of converting from one version of an object to another. These translation functions would be written by application programmers when class definitions are modified. The dossier driven type evolution system can then chain conversion functions to evolve from one version of an object to the next until the desired version has been computed. A mixture of the second and third techniques described above is being implemented for our POS.

9 Other Applications of Dossiers

Once a dossier generator is available several other applications become immediately apparent. Two of these applications are run-time type information and remote procedure call generation. There are essentially three options for using the proposed run-time type information feature[32] with dossiers. First, as Stroustrup suggests, the RTTI system can be queried to determine a type name which is then used as a key to access auxiliary information:

```
dossier_c *dp = lookup_dossier( typeid(*p).name() );
```

This has the obvious advantage that it uses only standard language features and is thus portable across all implementations.

Second, we could derive the `dossier_c` class from `Type_info` itself and cause `dynamic_cast<T>` and `typeid()` to return `dossier_c` instances. This would allow both the persistent object support library and applications to use extended type information directly through language supported mechanisms. Unfortunately, a preprocessor/support library approach to RTTI cannot be implemented portably owing to the variance in RTTI implementations. If the `dynamic_cast<T>` and `typeid` language features are implemented with support functions, then it would be possible to replace them with new versions returning references or pointers to dossiers. The dossier constructors could be enhanced to maintain any state in the base `Type_info` object required by the RTTI implementation. If, however, either of the RTTI constructs are implemented as inline code we see no mechanism, short of modifying the compiler, for substituting dossiers for `Type_info` objects.

The third technique would use a hybrid of the first two. The `Type_info` class could be extended with new (non-virtual) member functions (either through inheritance or direct modification) to support the functionality of dossiers. These member functions could use the type information in the `Type_info` object to access the dossier through a lookup table and return the appropriate values. Thus, to the user, it would appear that the `Type_info` object contained extended type information when, in fact, it did not. This approach has the advantage of simplicity and portability.

A dossier generator can also be used to build a remote procedure call (RPC) facility. One approach would be to enhance the generator itself to write RPC stubs which would be linked into the application. This would require parsing general function declarations (member and non-member) and possibly adding additional annotations for in, out, and in/out parameters. Our generator already performs this parsing. This implementation would render a powerful and convenient implementation of standard RPC. Another technique would be to implement a polymorphic RPC dispatcher capable of dynamically marshalling and unmarshalling arbitrary argument lists. This would allow advertising and accessing services dynamically and may be the basis for a CORBA-like object broker.

10 Current Status

The dossier generator, *goofie* (a General Object-Oriented Framework for Interface Expression), is largely complete. Goofie can generate dossiers for a large subset of C++ including all annotations described above. The omissions are due mainly to the highly decomposed nature of the Roskind grammar (i.e., rare or obscure grammar productions have not been fleshed out). An initial version of the polymorphic load and store code is complete (for a single platform) and is able to read and write objects and dossiers. The interface to the persistent store has been defined and two distinct stores have been implemented. The first uses a version of DBM supporting transaction semantics. The other converts objects to a serial byte stream for use across interprocess communication channels. We plan to add an interface to the EXODUS storage manager[4] shortly.

Although the design described here is quite general there are a number of limitations in the current system. Most important, we do not support pointers to the interior of objects (although the load store hooks allow crude handling of some cases). We also do not support unions or pointers to member functions in the current system. Only two styles of dynamically sized data members are

supported although many others can be envisioned. We are dissatisfied with the treatment of static data members mainly due to the uncertain semantics of persistent, shared members.

In terms of portability and simplicity of the solution there are several short comings. Of these, the most important is the requirement that the application programmer alter class definitions to include a reconstructor (optional), load/store hooks (optional), and the dossier accessor function (optional) or friend declaration. We see no solution to this problem given the initial problem constraints. Another problem is the possibility that the byte order and floating point format must be explicitly indicated in the dossier making it non-portable.

11 Future Work

The most important features currently unavailable in our system are heterogeneity and class evolution. To provide a universal and stable POS these are fundamental requirements. The design of these features is largely complete and an initial implementation should be completed soon. We hope to support both the simple static evolution algorithm used in [16] and the dynamic one described in Section 8. We are also investigating the ability to lazily load individual nodes of the object graph. Given our current implementation constraints this will probably require complete object encapsulation. In addition, dynamically loading class definitions in the form of dossiers and member functions is possible through the use of our object/meta-object server[19].

A portable, comprehensive dossier facility has applications in a variety of areas. Two applications related to our research are inter-language object transmission[17] and dynamic reconfiguration of software systems[5].

12 Related Work

Persistence for C++ systems has been the focus of vigorous and diverse research and development activity. Several commercial products, notably object-oriented database systems (e.g., [15]), provide persistence as a C++ extension. In addition, there are several experimental systems such as Arjuna which provide comprehensive support for persistent C++ objects.

Tables 1 and 2 summarize representative systems in terms of six distinguishing dimensions (see column headings). These correspond to important decisions which must be resolved by any persistent C++ system designer. We consider each in turn, offering a few clarifying comments. Further details are available from the references cited in each case.

Object description language: Several systems exploit C++ language extensions to describe persistent objects (Avalon, O++, OBST, SOS). Typically, these involve new key words or syntactic extensions. Arjuna and EC++ support a subset of full C++. For systems relying on persistent virtual memory (C**, E, and the Texas system), the C++ class definitions suffice for object description, though ObjectStore uses a database schema declaration facility for class evolution control. Similarly, the NIH class library, being ASCII file oriented, requires no object description language.

Dossier objects: Run-time information describing persistent objects is utilized by O++, ObjectStore, C**, and the Texas system. This information is captured in dossier objects in all but the Texas system, which uses a tabular representation. The remaining systems do not exploit dossier information.

Preprocessor use: Like the Utah approach, several systems use preprocessors to collect object description information. These include ObjectStore (optionally), OBST, Arjuna, Avalon, C**, EC++, and the Texas system. Three systems (E, SOS, and O++) rely on modified compilers.

Invocation of object storage and retrieval: A wide variety of techniques are relied upon for causing persistent objects to be saved and restored. The C++ option of overloaded new (i.e., placement syntax) is exploited by O++, ObjectStore, SOS and the Texas system. Reliance on a special base class conferring persistence is utilized by O++, SOS, Arjuna, Avalon, EC++, and the

System	Description Language	Dossiers	Preprocessor	Invocation	Implementation	Graph Traversal
Arjuna [8, 29]	Restricted C++	no	yes	special base class	rpc	no
Avalon [11]	Augmented C++	no	yes	special base class, `stable` keyword	rpc w/ transactions	inline code in r/w
C** [3, 18]	none	yes, not user visible	yes	object register method	vm and pointer swizzling	yes
E [20, 22, 21]	none	no	modified g++	parallel class hierarchy	vm and pointer swizzling	n/a
EC++ [25, 31]	Restricted C++	no	yes	named object, special base class	rpc	inline code in r/w
NIHCL [13]	none	no	no	special base class, r/w functions	ASCII files	inline code in r/w

Table 1: Summary of persistent objects systems and their approach.

System	Description Language	Dossiers	Preprocessor	Invocation	Implementation	Graph Traversal
O++ [12, 7, 1]	Augmented C++	yes	compiler	overloaded new, special base class	tagged byte stream	yes
ObjectStore [15]	DB schema	yes	yes	overloaded new	vm and pointer swizzling	no
OBST [6, 33, 24]	Augmented C++	no	yes	create in container object	copied on container commit	names as roots
SOS [27, 26, 28]	Augmented C++	no	modified g++	special base class, overloaded new	object fault on special pointer class	special pointer class
Texas [30, 34]	a.out	yes, packed in 2 tables	tdesc	overloaded new	vm and pointer swizzling	n/a?
Utah Dossiers	Augmented C++	yes, persistent objects	Roskind grammar-based (goofie)	r/w functions	distributed rpc w/ tagged byte stream	dossier driven

Table 2: Summary of persistent objects systems and their approach (Continued).

NIH class library. OBST, C**, Avalon, and E support involves keywords, object registration or parallel class. Like the Utah approach, the NIH class library provides explicit object read and write operations.

Implementation of storage and retrieval services: A wide variety of approaches are employed for implementing object dereferencing, copying, sharing, and inter-process transmission. Seamless pointer swizzling by page faulting is a principal advantage of persistent virtual memory based systems (ObjectStore, C** and E). Other systems rely on distributed processing, with special RPC-based services such as object identifier creation, binding and dereferencing. SOS uses a special persistent object pointer class, with faulting semantics. Systems providing transaction semantics include ObjectStore, OBST, and Avalon.

Transitive closure of object storage and retrieval: Finally, systems differ on whether object save operations include saving all referenced objects, i.e. saving object graphs, rather than individual objects. The point is moot for persistent virtual memory systems such as ObjectStore and C**. Other systems use special pointers, or named roots, to control save transitivity. Inline code controlling read/write depth is utilized by Avalon, EC++ and the NIH class library.

13 Conclusions

Using dossiers as the foundation for a persistent object store we have built a flexible, portable storage facility capable of supporting class evolution and platform heterogeneity. The requirements of the facility are such that any compiler compliant with the proposed ANSI C++ standard can be used to build applications with persistent objects. Our dossier generator, *goofie*, requires minimal alteration of application class descriptions and can be used where library source code is not available. In particular, the burden on the application programmer can be summarized as:

- pointers to dynamically sized memory must be annotated;
- a reconstructor must be added to the class or the default constructor must not have unwanted side effects;
- load/store hooks must be written for objects whose data values are application dependent; and,
- a virtual `__get_dossier` function should be added to a class, or the non-virtual `__get_dossier` function must be made a friend, or the class's data members must be publicly readable.

In many interesting classes the actual source code change is the addition of a friend declaration to allow access by the `__get_dossier` function.

The ability to apply this persistent store to large, existing software systems is an important aspect of our design and implementation. The dossiers generated for application objects can also be accessed from the proposed run-time type information system and can be used by the programmer to build application specific polymorphic functions. The ability to manipulate objects polymorphically allows us to serialize arbitrary object graphs and restore them providing the basis for inter-process object transmission and RPC stub generation. A prototype of the dossier generator, polymorphic I/O code, and object store are complete and work is continuing to enhance their functionality.

14 Acknowledgements

We gratefully acknowledge the contributions of the members of the Mach Shared Objects project. In particular, we would like to thank Mark Swanson, Jay Lepreau, and Doug Orr whose insight and assistance made this work possible. We would also like to thank the members of the Alpha_1 project who gave us their cooperation, support and creativity, especially Beth Cobb, Tim Mueller, Russ Fish, and Mark Bloomenthal.

15 Availability

The software described in this paper is available through anonymous ftp from `ftp.cs.utah.edu`. The distribution is a Unix compressed tar file, `pub/goofie.tar.Z`. This paper is included in the distribution. The software and paper are also available from the World Wide Web under the URL `http://www.cs.utah.edu/projects/mso/goofie/goofie.html`.

References

[1] Rakesh Agrawal, Shaul Dar, and Narain H. Gehani. The O++ database programming language: Implementation and experience. In *Proceedings of the IEEE 9th International Conference on Data Engineering*. IEEE Computer Press, 1993.

[2] Alpha_1 Project. Integrated computer aided design and manufacturing: An overview of Alpha_1. Technical report, University of Utah, Dept. of Computer Science, March 5, 1992.

[3] Vinny Cahill, Chris Horn, Andre Kramer, Maurice Martin, and Gradimir Starovic. C** and Eiffel**: Languages for distribution and persistence. In *Proceedings of the 1990 OSF Microkernel Applications Workshop*, Grenoble, France, 1990.

[4] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage management for objects in EXODUS. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 341–369. Addison-Wesley, 1989.

[5] John B. Carter, Bryan Ford, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: A tool for building efficient and flexible systems. In *Proc. Fourth Workshop on Workstation Operating Systems*, October 1993.

[6] Eduardo Casais, Michael Ranft, Bernhard Schiefer, Dietmar Theobald, and Walter Zimmer. OBST — An overview. Technical report, Forschungszentrum Informatik (FZI), D-76131 Karlsruhe, Germany, 1993.

[7] S. Dar, N. H. Gehani, and H. V. Jagadish. CQL++: A SQL for a C++ based object-oriented DBMS. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology — EDBT '92: Proceedings of the 3rd International Conference on Extending Database Technology*, Vienna, Austria, March, 1992, 1992. Springer-Verlag.

[8] G.N. Dixon, G.D. Parrington, S.K. Shrivastava, and S.M. Wheater. The treatment of persistent objects in Arjuna. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 169–189, University of Nottingham, July 10–14, 1989. Cambridge University Press.

[9] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *USENIX C++ Conference Proceedings*, pages 1–20, Portland, Oregon, August 1992. The USENIX Association.

[10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[11] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Data Management Systems. Morgan Kaufmann Publishers, Menlo Park, CA, 1991.

[12] N. H. Gehani. OdeFS: A file system interface to an object-oriented database. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1989.

[13] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++.* John Wiley & Sons, 1990.

[14] John A. Interrante and Mark A. Linton. Runtime access to type information in C++. In *USENIX Proceedings C++ Conference*, pages 233–240. USENIX Association, 1990.

[15] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[16] Robert W. Mecklenburg. The specification for a binary file format for Alpha_1 models. Alpha_1 technical report 88-6, University of Utah, 1988.

[17] Robert W. Mecklenburg. *Towards a Language Independent Object System.* PhD thesis, University of Utah, Salt Lake City, Utah, June 1991.

[18] Michael Mock, Reinhold Kroeger, and Vinny Cahill. Implementing atomic objects with the RelaX transaction facility. *Computing Systems*, 5(3):259–304, Summer 1992.

[19] Douglas B. Orr and Robert W. Mecklenburg. OMOS — An object server for program execution. In *Proc. International Workshop on Object Oriented Operating Systems*, pages 200–209, Paris, September 1992. IEEE Computer Society. Also available as technical report UUCS-92-033.

[20] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software—Practice and Experience*, 19(12):1115–1150, December 1989.

[21] Joel E. Richardson and Michael J. Carey. Implementing persistence in E. In John Rosenberg and David Koch, editors, *Persistent Object Systems: Proceedings of the Third International Workshop*, Workshops in Computing, pages 175–199. Springer-Verlag, Newcastle, Australia, January 10–13, 1989, 1990.

[22] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. Technical Report 814, Computer Science Department, University of Wisconsin, Madison, WI, February 1989.

[23] Jim Roskind. A yacc-able C++ 2.1 grammar, and the resulting ambiguities. July 1991.

[24] Bernhard Schiefer, Dietmar Theobald, and Jürgen Uhl. User's guide: OBST release 3.3. Technical report, Forschungszentrum Informatik (FZI), D-76131 Karlsruhe, Germany, July 1993.

[25] Manuel Sequeira and José Alves Marques. Can C++ be used for programming distributed and persistent objects? In *Proceedings 1991 International Workshop on Object Orientation in Operating Systems*, pages 173–176, Palo Alto, CA, October 17–18, 1991. IEEE Computer Society Press.

[26] Marc Shapiro. Prototyping a distributed object-oriented operating system on Unix. In *Proceedings of the First USENIX/SERC Workshop on Experiences with Distributed and Multiprocesor Systems*, pages 311–331, Fort Lauderdale, FL, October 5–6, 1989. Usenix Association.

[27] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating systems—Assessment and perspectives. *Computing Systems*, 2(4):287–337, Fall 1989.

[28] Marc Shapiro and Laurence Mosseri. A simple object storage system. In John Rosenberg and David Koch, editors, *Persistent Object Systems: Proceedings of the Third International Workshop*, Workshops in Computing, pages 272–276. Springer-Verlag, Newcastle, Australia, January 10–13, 1989, 1990.

[29] Santosh K. Shrivastava et al. *The Arjuna System Programmer's Guide.* Arjuna Research Group, Computing Laboratory, University of Newcastle upon Tyne, UK, February 1992. Public Release 1.0.

[30] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of The Fifth International Workshop on Persistent Object Systems (POS-V)*, San Miniato, Italy, September, 1992, 1992.

[31] Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, José Pereira, Paulo Guedes, and José Alves Marques. Distribution and persistence in the IK platform: Overview an d evaluation. *Computing Systems*, 6(4):391–424, Fall 1993.

[32] Bjarne Stroustrup and Dmitry Lenkov. Run-time type identification for C++ (revised). In *USENIX C++ Conference Proceedings*, pages 313–339, Portland, Oregon, August 1992. The USENIX Association.

[33] Jürgen Uhl, Dietmar Theobald, Bernhard Schiefer, Michael Ranft, Walter Zimmer, and Jochen Alt. The object management system of STONE: OBST release 3.3. Technical report, Forschungszentrum Informatik (FZI), D-76131 Karlsruhe, Germany, July 1993.

[34] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 364–377, Dourdan, France, September 24–25, 1992. IEEE Computer Society Press.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

* sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
* fostering innovation and communicating both research and technological developments,
* providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with the MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association, devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well. There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided.

USENIX Association membership services include:

* Subscription to *;login:*, a bi-monthly newsletter;
* Subscription to *Computing Systems*, a refereed technical quarterly;
* Discounts on various UNIX and technical publications available for purchase;
* Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
* The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
* The right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association:

ANDATACO
ASANTÉ Technologies, Inc.
Frame Technology Corporation
Network Computing Devices, Inc.
OTA Limited Partnership
Quality Micro Systems, Inc.
UUNET Technologies, Inc.
Enterprise System Management Corporations (SAGE supporting members)

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Email: *office@usenix.org*
Phone: +1-510-528-8649
Fax: +1-510-548-5738

ISBN 1-880446-60-X